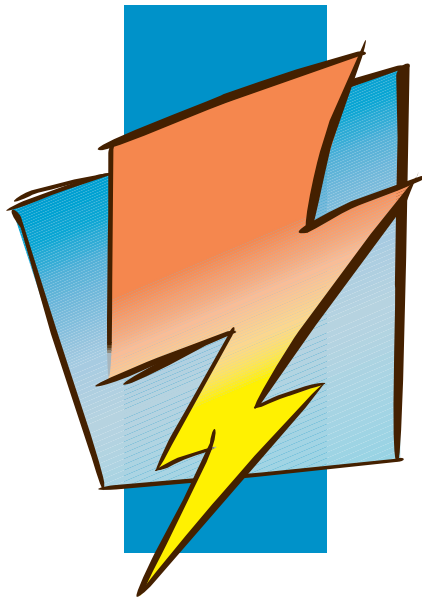


*Open Watcom C++*  
*Class Library Reference*



*First Edition*

Open **Watcom**

## ***Notice of Copyright***

Copyright © 2002-2006 the Open Watcom Contributors. Portions Copyright © 1984-2002 Sybase, Inc. and its subsidiaries. All rights reserved.

Any part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of anyone.

For more information please visit <http://www.openwatcom.org/>

Printed in U.S.A.

---

# **Preface**

Open Watcom C++ is an implementation of the C++ programming language. In addition to the C++ draft standard, the compiler supports numerous extensions for the PC environment.

This manual describes the Open Watcom C++ Class Libraries for DOS, Windows 3.x, Windows NT, Windows 95, 16-bit OS/2 1.x, 32-bit OS/2, and QNX. It includes a String Class, a Complex Class, Container Classes, and an I/O Stream hierarchy of classes. The Container classes include a set of intrusive, value and pointer list classes with their associated iterators.

This book was produced with the Open Watcom GML electronic publishing system, a software tool developed by WATCOM. In this system, writers use an ASCII text editor to create source files containing text annotated with tags. These tags label the structural elements of the document, such as chapters, sections, paragraphs, and lists. The Open Watcom GML software, which runs on a variety of operating systems, interprets the tags to format the text into a form such as you see here. Writers can produce output for a variety of printers, including laser printers, using separately specified layout directives for such things as font selection, column width and height, number of columns, etc. The result is type-set quality copy containing integrated text and graphics.

July, 1997.

## ***Trademarks Used in this Manual***

IBM is a registered trademark and OS/2 is a trademark of International Business Machines Corp.

Microsoft is a registered trademark of Microsoft Corp. Windows, Windows NT and Windows 95 are trademarks of Microsoft Corp.

QNX is a registered trademark of QNX Software Systems Ltd.

WATCOM is a trademark of Sybase, Inc. and its subsidiaries.



# Table of Contents

Open Watcom C++ Class Library Reference .....	1
1 Header Files .....	3
2 Common Types .....	9
3 Predefined Objects .....	11
3.1 cin .....	11
3.2 cout .....	11
3.3 cerr .....	12
3.4 clog .....	12
4 istream Input .....	13
4.1 Formatted Input: Extractors .....	13
4.2 Unformatted Input .....	13
5 ostream Output .....	15
5.1 Formatted Output: Inserters .....	15
5.2 Unformatted Output .....	15
6 Library Functions and Types .....	17
7 Complex Class .....	19
Complex Class Description .....	20
Complex abs() .....	23
Complex acos() .....	24
Complex acosh() .....	25
Complex arg() .....	26
Complex asin() .....	27
Complex asinh() .....	28
Complex atan() .....	29
Complex atanh() .....	30
Complex() .....	31
Complex() .....	32
Complex() .....	33
~Complex() .....	34
Complex conj() .....	35
Complex cos() .....	36
Complex cosh() .....	37
Complex exp() .....	38
imag() .....	39
Complex imag() .....	40

# Table of Contents

Complex log()	41
Complex log10()	42
Complex norm()	43
Complex operator !=()	44
Complex operator *()	45
operator *=( )	46
operator +( )	47
Complex operator +( )	48
operator +=( )	49
operator -( )	50
Complex operator -( )	51
operator -=( )	52
Complex operator /( )	53
operator /=( )	54
Complex operator <<( )	55
operator =( )	56
Complex operator ==( )	57
Complex operator >>( )	58
Complex polar()	59
Complex pow()	60
real()	61
Complex real()	62
Complex sin()	63
Complex sinh()	64
Complex sqrt()	65
Complex tan()	66
Complex tanh()	67
8 Container Exception Classes	69
WCEXCEPT Class Description	70
WCEXCEPT()	71
~WCEXCEPT()	72
exceptions()	73
wc_state	74
WCITEREXCEPT Class Description	76
WCITEREXCEPT()	77
~WCITEREXCEPT()	78
exceptions()	79
wciter_state	80
9 Container Allocators and Deallocators	81

# Table of Contents

10 Hash Containers .....	87
WCPtrHashDict<Key, Value> Class Description .....	88
WCPtrHashDict() .....	90
WCPtrHashDict() .....	91
WCPtrHashDict() .....	92
~WCPtrHashDict() .....	93
bitHash() .....	94
buckets() .....	95
clear() .....	96
clearAndDestroy() .....	97
contains() .....	98
entries() .....	99
find() .....	100
findKeyAndValue() .....	101
forAll() .....	102
insert() .....	103
isEmpty() .....	104
operator []() .....	105
operator []() .....	106
operator =() .....	107
operator ==() .....	108
remove() .....	109
resize() .....	110
WCPtrHashTable<Type>, WCPtrHashSet<Type> Class Description .....	111
WCPtrHashSet() .....	113
WCPtrHashSet() .....	114
WCPtrHashSet() .....	115
~WCPtrHashSet() .....	116
WCPtrHashTable() .....	117
WCPtrHashTable() .....	118
WCPtrHashTable() .....	119
~WCPtrHashTable() .....	120
bitHash() .....	121
buckets() .....	122
clear() .....	123
clearAndDestroy() .....	124
contains() .....	125
entries() .....	126
find() .....	127
forAll() .....	128
insert() .....	129
isEmpty() .....	130

# Table of Contents

occurrencesOf()	131
operator =()	132
operator ==()	133
remove()	134
removeAll()	135
resize()	136
WCValHashDict<Key, Value> Class Description	137
WCValHashDict()	139
WCValHashDict()	140
WCValHashDict()	141
~WCValHashDict()	142
bitHash()	143
buckets()	144
clear()	145
contains()	146
entries()	147
find()	148
findKeyAndValue()	149
forAll()	150
insert()	151
isEmpty()	152
operator []()	153
operator []()	154
operator =()	155
operator ==()	156
remove()	157
resize()	158
WCValHashTable<Type>, WCValHashSet<Type> Class Description	159
WCValHashSet()	161
WCValHashSet()	162
WCValHashSet()	163
~WCValHashSet()	164
WCValHashTable()	165
WCValHashTable()	166
WCValHashTable()	167
~WCValHashTable()	168
bitHash()	169
buckets()	170
clear()	171
contains()	172
entries()	173
find()	174



# Table of Contents

forAll()	175
insert()	176
isEmpty()	177
occurrencesOf()	178
operator =()	179
operator ==()	180
remove()	181
removeAll()	182
resize()	183
11 Hash Iterators	185
WCPtrHashDictIter<Key, Value> Class Description	186
WCPtrHashDictIter()	187
WCPtrHashDictIter()	188
~WCPtrHashDictIter()	189
container()	190
key()	191
operator ()()	192
operator ++()	193
reset()	194
reset()	195
value()	196
WCValHashDictIter<Key, Value> Class Description	197
WCValHashDictIter()	198
WCValHashDictIter()	199
~WCValHashDictIter()	200
container()	201
key()	202
operator ()()	203
operator ++()	204
reset()	205
reset()	206
value()	207
WCPtrHashSetIter<Type>, WCPtrHashTableIter<Type> Class Description	208
WCPtrHashSetIter()	209
WCPtrHashSetIter()	210
~WCPtrHashSetIter()	211
WCPtrHashTableIter()	212
WCPtrHashTableIter()	213
~WCPtrHashTableIter()	214
container()	215
current()	216

# Table of Contents

operator ()()	217
operator ++()	218
reset()	219
reset()	220
WCValHashSetIter<Type>, WCValHashTableIter<Type> Class Description	221
WCValHashSetIter()	222
WCValHashSetIter()	223
~WCValHashSetIter()	224
WCValHashTableIter()	225
WCValHashTableIter()	226
~WCValHashTableIter()	227
container()	228
current()	229
operator ()()	230
operator ++()	231
reset()	232
reset()	233
12 List Containers	235
WCDLink Class Description	237
WCDLink()	238
~WCDLink()	239
WCIsvSList<Type>, WCIsvDList<Type> Class Description	240
WCIsvSList()	243
WCIsvSList()	244
~WCIsvSList()	245
WCIsvDList()	246
WCIsvDList()	247
~WCIsvDList()	248
append()	249
clear()	250
clearAndDestroy()	251
contains()	252
entries()	253
find()	254
findLast()	255
forAll()	256
get()	257
index()	258
index()	259
insert()	260
isEmpty()	261

# Table of Contents

operator =()	262
operator ==()	263
WCPtrSList<Type>, WCPtrDList<Type> Class Description	264
WCPtrSList()	266
WCPtrSList()	267
WCPtrSList()	268
~WCPtrSList()	269
WCPtrDList()	270
WCPtrDList()	271
WCPtrDList()	272
~WCPtrDList()	273
append()	274
clear()	275
clearAndDestroy()	276
contains()	277
entries()	278
find()	279
findLast()	280
forAll()	281
get()	282
index()	283
insert()	284
isEmpty()	285
operator =()	286
operator ==()	287
WCSLink Class Description	288
WCSLink()	289
~WCSLink()	290
WCValSList<Type>, WCValDList<Type> Class Description	291
WCValSList()	294
WCValSList()	295
WCValSList()	296
~WCValSList()	297
WCValDList()	298
WCValDList()	299
WCValDList()	300
~WCValDList()	301
append()	302
clear()	303
clearAndDestroy()	304
contains()	305
entries()	306

# Table of Contents

find()	307
findLast()	308
forAll()	309
get()	310
index()	311
insert()	312
isEmpty()	313
operator =()	314
operator ==()	315
13 List Iterators	317
WCIsvConstSListIter<Type>, WCIsvConstDListIter<Type> Class Description	318
WCIsvConstSListIter()	320
WCIsvConstSListIter()	321
~WCIsvConstSListIter()	322
WCIsvConstDListIter()	323
WCIsvConstDListIter()	324
~WCIsvConstDListIter()	325
container()	326
current()	327
operator ()()	328
operator ++()	329
operator +=()	330
operator --()	331
operator -=()	332
reset()	333
reset()	334
WCIsvSListIter<Type>, WCIsvDListIter<Type> Class Description	335
WCIsvSListIter()	337
WCIsvSListIter()	338
~WCIsvSListIter()	339
WCIsvDListIter()	340
WCIsvDListIter()	341
~WCIsvDListIter()	342
append()	343
container()	344
current()	345
insert()	346
operator ()()	347
operator ++()	348
operator +=()	349
operator --()	350

# Table of Contents

operator -=()	351
reset()	352
reset()	353
WCPtrConstSListIter<Type>, WCPtrConstDListIter<Type> Class Description	354
WCPtrConstSListIter()	356
WCPtrConstSListIter()	357
~WCPtrConstSListIter()	358
WCPtrConstDListIter()	359
WCPtrConstDListIter()	360
~WCPtrConstDListIter()	361
container()	362
current()	363
operator ()()	364
operator ++()	365
operator +=()	366
operator --()	367
operator -=()	368
reset()	369
reset()	370
WCPtrSListIter<Type>, WCPtrDListIter<Type> Class Description	371
WCPtrSListIter()	373
WCPtrSListIter()	374
~WCPtrSListIter()	375
WCPtrDListIter()	376
WCPtrDListIter()	377
~WCPtrDListIter()	378
append()	379
container()	380
current()	381
insert()	382
operator ()()	383
operator ++()	384
operator +=()	385
operator --()	386
operator -=()	387
reset()	388
reset()	389
WCValConstSListIter<Type>, WCValConstDListIter<Type> Class Description	390
WCValConstSListIter()	392
WCValConstSListIter()	393
~WCValConstSListIter()	394
WCValConstDListIter()	395

# Table of Contents

WCValConstDListIter() .....	396
~WCValConstDListIter() .....	397
container() .....	398
current() .....	399
operator ()() .....	400
operator ++() .....	401
operator +=() .....	402
operator --() .....	403
operator -=() .....	404
reset() .....	405
reset() .....	406
WCValSListIter<Type>, WCValDListIter<Type> Class Description .....	407
WCValSListIter() .....	410
WCValSListIter() .....	411
~WCValSListIter() .....	412
WCValDListIter() .....	413
WCValDListIter() .....	414
~WCValDListIter() .....	415
append() .....	416
container() .....	417
current() .....	418
insert() .....	419
operator ()() .....	420
operator ++() .....	421
operator +=() .....	422
operator --() .....	423
operator -=() .....	424
reset() .....	425
reset() .....	426
14 Queue Container .....	427
WCQueue<Type,FType> Class Description .....	428
WCQueue() .....	430
WCQueue() .....	431
~WCQueue() .....	432
clear() .....	433
entries() .....	434
first() .....	435
get() .....	436
insert() .....	437
isEmpty() .....	438
last() .....	439

# Table of Contents

15 Skip List Containers .....	441
WCPtrSkipListDict<Key, Value> Class Description .....	442
WCPtrSkipListDict() .....	444
WCPtrSkipListDict() .....	445
WCPtrSkipListDict() .....	446
~WCPtrSkipListDict() .....	447
clear() .....	448
clearAndDestroy() .....	449
contains() .....	450
entries() .....	451
find() .....	452
findKeyAndValue() .....	453
forAll() .....	454
insert() .....	455
isEmpty() .....	456
operator []() .....	457
operator []() .....	458
operator =() .....	459
operator ==() .....	460
remove() .....	461
WCPtrSkipList<Type>, WCPtrSkipListSet<Type> Class Description .....	462
WCPtrSkipListSet() .....	464
WCPtrSkipListSet() .....	465
WCPtrSkipListSet() .....	466
~WCPtrSkipListSet() .....	467
WCPtrSkipList() .....	468
WCPtrSkipList() .....	469
WCPtrSkipList() .....	470
~WCPtrSkipList() .....	471
clear() .....	472
clearAndDestroy() .....	473
contains() .....	474
entries() .....	475
find() .....	476
forAll() .....	477
insert() .....	478
isEmpty() .....	479
occurrencesOf() .....	480
operator =() .....	481
operator ==() .....	482
remove() .....	483
removeAll() .....	484

# Table of Contents

WCValSkipListDict<Key,Value> Class Description .....	485
WCValSkipListDict() .....	487
WCValSkipListDict() .....	488
WCValSkipListDict() .....	489
~WCValSkipListDict() .....	490
clear() .....	491
contains() .....	492
entries() .....	493
find() .....	494
findKeyAndValue() .....	495
forAll() .....	496
insert() .....	497
isEmpty() .....	498
operator []() .....	499
operator []() .....	500
operator =() .....	501
operator ==() .....	502
remove() .....	503
WCValSkipList<Type>, WCValSkipListSet<Type> Class Description .....	504
WCValSkipListSet() .....	506
WCValSkipListSet() .....	507
WCValSkipListSet() .....	508
~WCValSkipListSet() .....	509
WCValSkipList() .....	510
WCValSkipList() .....	511
WCValSkipList() .....	512
~WCValSkipList() .....	513
clear() .....	514
contains() .....	515
entries() .....	516
find() .....	517
forAll() .....	518
insert() .....	519
isEmpty() .....	520
occurrencesOf() .....	521
operator =() .....	522
operator ==() .....	523
remove() .....	524
removeAll() .....	525
16 Stack Container .....	527
WCStack<Type,FType> Class Description .....	528



# Table of Contents

WCStack()	530
WCStack()	531
~WCStack()	532
clear()	533
entries()	534
isEmpty()	535
pop()	536
push()	537
top()	538
17 Vector Containers	539
WCPtrSortedVector<Type>, WCPtrOrderedVector<Type> Class Description	540
WCPtrOrderedVector()	543
WCPtrOrderedVector()	544
~WCPtrOrderedVector()	545
WCPtrSortedVector()	546
WCPtrSortedVector()	547
~WCPtrSortedVector()	548
append()	549
clear()	550
clearAndDestroy()	551
contains()	552
entries()	553
find()	554
first()	555
index()	556
insert()	557
insertAt()	558
isEmpty()	559
last()	560
occurrencesOf()	561
operator []()	562
operator =()	563
operator ==()	564
prepend()	565
remove()	566
removeAll()	567
removeAt()	568
removeFirst()	569
removeLast()	570
resize()	571
WCPtrVector<Type> Class Description	572

# Table of Contents

WCPtrVector() .....	573
WCPtrVector() .....	574
WCPtrVector() .....	575
~WCPtrVector() .....	576
clear() .....	577
clearAndDestroy() .....	578
length() .....	579
operator []() .....	580
operator =() .....	581
operator ==() .....	582
resize() .....	583
WCValSortedVector<Type>, WCValOrderedVector<Type> Class Description .....	584
WCValOrderedVector() .....	587
WCValOrderedVector() .....	588
~WCValOrderedVector() .....	589
WCValSortedVector() .....	590
WCValSortedVector() .....	591
~WCValSortedVector() .....	592
append() .....	593
clear() .....	594
contains() .....	595
entries() .....	596
find() .....	597
first() .....	598
index() .....	599
insert() .....	600
insertAt() .....	601
isEmpty() .....	602
last() .....	603
occurrencesOf() .....	604
operator []() .....	605
operator =() .....	606
operator ==() .....	607
prepend() .....	608
remove() .....	609
removeAll() .....	610
removeAt() .....	611
removeFirst() .....	612
removeLast() .....	613
resize() .....	614
WCValVector<Type> Class Description .....	615
WCValVector() .....	617

# Table of Contents

WCValVector() .....	618
WCValVector() .....	619
~WCValVector() .....	620
clear() .....	621
length() .....	622
operator []() .....	623
operator =() .....	624
operator ==() .....	625
resize() .....	626
<b>18 Input/Output Classes .....</b>	<b>627</b>
<b>filebuf Class Description .....</b>	<b>628</b>
attach() .....	630
close() .....	631
fd() .....	632
filebuf() .....	633
filebuf() .....	634
filebuf() .....	635
~filebuf() .....	636
is_open() .....	637
open() .....	638
openprot .....	639
overflow() .....	640
pbackfail() .....	641
seekoff() .....	642
setbuf() .....	643
sync() .....	644
underflow() .....	645
<b>fstream Class Description .....</b>	<b>646</b>
fstream() .....	647
fstream() .....	648
fstream() .....	649
fstream() .....	650
~fstream() .....	651
open() .....	652
<b>fstreambase Class Description .....</b>	<b>653</b>
attach() .....	654
close() .....	655
fstreambase() .....	656
fstreambase() .....	657
fstreambase() .....	658
fstreambase() .....	659

# Table of Contents

~fstreambase() .....	660
is_open() .....	661
fd() .....	662
open() .....	663
rdbuf() .....	664
setbuf() .....	665
ifstream Class Description .....	666
ifstream() .....	667
ifstream() .....	668
ifstream() .....	669
ifstream() .....	670
~ifstream() .....	671
open() .....	672
ios Class Description .....	673
bad() .....	675
bitalloc() .....	676
clear() .....	677
eof() .....	678
exceptions() .....	679
fail() .....	680
fill() .....	681
flags() .....	682
fmtflags .....	683
good() .....	687
init() .....	688
ios() .....	689
ios() .....	690
~ios() .....	691
iostate .....	692
iword() .....	693
openmode .....	694
operator !() .....	696
operator void *() .....	697
precision() .....	698
pword() .....	699
rdbuf() .....	700
rdstate() .....	701
seekdir .....	702
setf() .....	703
setstate() .....	704
sync_with_stdio() .....	705
tie() .....	706

# Table of Contents

unsetf() .....	707
width() .....	708
xalloc() .....	709
iostream Class Description .....	710
iostream() .....	711
iostream() .....	712
iostream() .....	713
~iostream() .....	714
operator =() .....	715
operator =() .....	716
istream Class Description .....	717
eatwhite() .....	719
gcount() .....	720
get() .....	721
get() .....	722
get() .....	723
get() .....	724
getline() .....	725
ignore() .....	726
ipfx() .....	727
isfx() .....	728
istream() .....	729
istream() .....	730
istream() .....	731
~istream() .....	732
operator =() .....	733
operator =() .....	734
operator >>() .....	735
operator >>() .....	736
operator >>() .....	737
operator >>() .....	738
operator >>() .....	739
operator >>() .....	740
peek() .....	741
putback() .....	742
read() .....	743
seekg() .....	744
seekg() .....	745
sync() .....	746
tellg() .....	747
istream Class Description .....	748
istream() .....	749

# Table of Contents

istream()	750
~istream()	751
Manipulators	752
manipulator dec()	753
manipulator endl()	754
manipulator ends()	755
manipulator flush()	756
manipulator hex()	757
manipulator oct()	758
manipulator resetiosflags()	759
manipulator setbase()	760
manipulator setfill()	761
manipulator setiosflags()	762
manipulator setprecision()	763
manipulator setw()	764
manipulator setwidth()	765
manipulator ws()	766
ofstream Class Description	767
ofstream()	768
ofstream()	769
ofstream()	770
ofstream()	771
~ofstream()	772
open()	773
ostream Class Description	774
flush()	776
operator <<()	777
operator <<()	778
operator <<()	779
operator <<()	781
operator <<()	782
operator <<()	783
operator <<()	784
operator =()	785
operator =()	786
opfx()	787
osfx()	788
ostream()	789
ostream()	790
ostream()	791
~ostream()	792
put()	793

# Table of Contents

seekp() .....	794
seekp() .....	795
tellp() .....	796
write() .....	797
ostream Class Description .....	798
ostream() .....	799
ostream() .....	800
~ostream() .....	801
pcount() .....	802
str() .....	803
stdiobuf Class Description .....	804
overflow() .....	805
stdiobuf() .....	806
stdiobuf() .....	807
~stdiobuf() .....	808
sync() .....	809
underflow() .....	810
streambuf Class Description .....	811
allocate() .....	815
base() .....	816
blen() .....	817
dbp() .....	818
do_sgetn() .....	819
do_sputn() .....	820
doallocate() .....	821
eback() .....	822
ebuf() .....	823
egptr() .....	824
epptr() .....	825
gbump() .....	826
gptr() .....	827
in_avail() .....	828
out_waiting() .....	829
overflow() .....	830
pbackfail() .....	831
pbase() .....	832
pbump() .....	833
pptr() .....	834
sbumpc() .....	835
seekoff() .....	836
seekpos() .....	837
setb() .....	838

# Table of Contents

setbuf()	839
setg()	840
setp()	841
sgetc()	842
sgetchar()	843
sgetn()	844
snextc()	845
speekc()	846
sputbackc()	847
sputc()	848
sputn()	849
stossc()	850
streambuf()	851
streambuf()	852
~streambuf()	853
sync()	854
unbuffered()	855
underflow()	856
strstream Class Description	857
str()	858
strstream()	859
strstream()	860
~strstream()	861
strstreambase Class Description	862
rdbuf()	863
strstreambase()	864
strstreambase()	865
~strstreambase()	866
strstreambuf Class Description	867
alloc_size_increment()	869
doallocate()	870
freeze()	871
overflow()	872
seekoff()	873
setbuf()	874
str()	875
strstreambuf()	876
strstreambuf()	877
strstreambuf()	878
strstreambuf()	879
~strstreambuf()	881
sync()	882



# Table of Contents

underflow() .....	883
19 String Class .....	885
String Class Description .....	886
alloc_mult_size() .....	888
get_at() .....	889
index() .....	890
length() .....	891
lower() .....	892
match() .....	893
operator !() .....	894
String operator !=() .....	895
operator ()() .....	896
operator () .....	897
String operator +() .....	898
operator +=() .....	899
String operator <() .....	900
String operator <<() .....	901
String operator <=() .....	902
operator =() .....	903
String operator ==() .....	904
String operator >() .....	905
String operator >=() .....	906
String operator >>() .....	907
operator []() .....	908
operator char() .....	909
operator char const *() .....	910
put_at() .....	911
String() .....	912
String() .....	913
String() .....	914
String() .....	915
String() .....	916
~String() .....	917
upper() .....	918
String valid() .....	919
valid() .....	920



***Open Watcom C++ Class Library  
Reference***



---

# 1 Header Files

The following header files are supplied with the Open Watcom C++ library. When a class or function from the library is used in a source file the related header file should be included in that source file. The header files can be included multiple times and in any order with no ill effect.

The facilities of the C standard library can be used in C++ programs by including the appropriate "cname" header. In that case all of the C standard library functions are in namespace `std`. For example, to use function `std::printf` one should include the header `cstdio`. Note that the cname headers declare in the global namespace any non-standard names they contain as extensions. It is also possible to include in a C++ program the same headers used by C programs. In that case, the standard functions are in both the global namespace as well as in namespace `std`.

Some of C++ standard library headers described below come in a form with a `.h` extension and in a form without an extension. The extensionless headers declare their library classes and functions in namespace `std`. The headers with a `.h` extension declare their library classes and functions in both the global namespace and in namespace `std`. Such headers are provided as a convenience and for compatibility with legacy code. Programs that intend to conform to Standard C++ should use the extensionless headers to access the facilities of the C++ standard library.

Certain headers defined by Standard C++ have names that are longer than the 8.3 limit imposed by the FAT16 filesystem. Such headers are provided with names that are truncated to eight characters so they can be used with the DOS host. However, one can still refer to them in `#include` directives using their full names as defined by the standard. If the Open Watcom C++ compiler is unable to open a header with the long name, it will truncate the name and try again.

The Open Watcom C++ library contains some components that were developed before C++ was standardized. These legacy components continue to be supported and are described in this documentation.

The header files are all located in the `\WATCOM\H` directory.

**algorithm (algorithm)** This header file defines the standard algorithm templates.

- complex** This header file defines the `std::complex` class template and related function templates. This template can be instantiated for the three different floating point types. It can be used to represent complex numbers and to perform complex arithmetic.
- complex.h** This header file defines the legacy `Complex` class. This class is used to represent complex numbers and to perform complex arithmetic. The class defined in this header is not the Standard C++ `std::complex` class template.
- exception/exception.h (exceptio/exceptio.h)** This header file defines components to be used with the exception handling mechanism. It defines the base class of the standard exception hierarchy.
- functional (function)** This header file defines the standard functional templates. This includes the functors and binders described by Standard C++.
- fstream/fstream.h** This header file defines the `filebuf`, `fstreambase`, `ifstream`, `ofstream`, and `fstream` classes. These classes are used to perform C++ file input and output operations. The various class members are declared and inline member functions for the classes are defined.
- generic.h** This header file is part of the macro support required to implement generic containers prior to the introduction of templates in the C++ language. It is retained for backwards compatibility.
- iomanip/iomanip.h** This header file defines the parameterized manipulators.
- ios/ios.h** This header file defines the class `ios` that is used as a base of the other `iostream` classes.
- iosfwd/iosfwd.h** This header file provides forward declarations of the `iostream` classes. It should be used in cases where the full class definitions are not needed but where one still wants to declare pointers or references to `iostream` related objects. Typically this occurs in a header for another class that wants to provide overloaded inserter or extractor operators. By including `iosfwd` instead of `iostream` (for example), compilation speed can be improved because less material must be processed by the compiler.
- Note that including `iosfwd` is the only appropriate way to forward declare the `iostream` classes. Manually writing forward declarations is not recommended.
- iostream/iostream.h** This header file (indirectly) defines the `ios`, `istream`, `ostream`, and `iostream` classes. These classes form the basis of the C++ formatted input and output support. The various class members are declared and inline member

## 4 Header Files

functions for the classes are defined. The `cin`, `cout`, `cerr`, and `clog` predefined objects are declared along with the non-parameterized manipulators.

- istream/istream.h** This header file defines class `istream` and class `iostream`. It also defines their associated parameterless manipulators.
- iterator** This header file defines several templates to facilitate the handling of iterators. In particular, it defines the `std::iterator_traits` template as well as several other supporting iterator related templates.
- limits** This header file defines the `std::numeric_limits` template and provides specializations of that template for each of the built-in types.
- Note that this header is not directly related to the header `limits.h` from the C standard library (or to the C++ form of that header, `climits`).
- list** This header file defines the `std::list` class template. It provides a way to make a sequence of objects with efficient insert and erase operations.
- map** This header file defines the `std::map` and `std::multimap` class templates. They provide ways to associate keys to values.
- memory** This header file defines the default allocator template, `std::allocator`, as well as several function templates for manipulating raw (uninitialized) memory regions. In addition this header defines the `std::auto_ptr` template.
- Note that the header `memory.h` is part of the Open Watcom C library and is unrelated to `memory`.
- new/new.h** This header file provides declarations to be used with the intrinsic `operator new` and `operator delete` memory management functions.
- numeric** This header file defines several standard algorithm templates pertaining to numerical computation.
- ostream/ostream.h** This header file defines class `ostream`. It also defines its associated parameterless manipulators.
- set** This header file defines the `std::set` and `std::multiset` class templates. They provide ways to make ordered collections of objects with efficient insert, erase, and find operations.

- stdiobuf.h** This header file defines the `stdiobuf` class which provides the support for the C++ input and output operations to standard input, standard output, and standard error streams.
- streambuf/streambuf.h (streambu/streambu.h)** This header file defines the `streambuf` class which provides the support for buffering of input and output operations. This header file is automatically included by the `iostream.h` header file.
- string** This header file defines the `std::basic_string` class template. It also contains the type definitions for `std::string` and `std::wstring`. In addition, this header contains specializations of the `std::char_traits` template for both characters and wide characters.
- string.hpp** This header file defines the legacy `String` class. The `String` class is used to manipulate character strings. Note that the `hpp` extension is used to avoid colliding with the Standard C `string.h` header file. The class defined in this header is not the Standard C++ `std::string` class.
- strstream.h (strstrea.h)** This header files defines the `strstreambuf`, `strstreambase`, `istrstream`, `ostrstream`, and `strstream` classes. These classes are used to perform C++ in-memory formatting. The various class members are declared and inline member functions for the classes are defined.
- vector** This header contains the `std::vector` class template.
- wcdefs.h** This header file contains definitions used by the Open Watcom legacy container libraries. If a container class needs any of these definitions, the file is automatically included.
- Note that all headers having names that start with "wc" are related to the legacy container libraries.
- wcbase.h** This header file defines the base classes which are used by the list containers.
- wcocom.h** This header file defines the classes which are common to the list containers.
- wcibase.h** This header file defines the base classes which are used by the list iterators.
- wclist.h** This header file defines the `list` container classes. The available list container classes are single and double linked versions of intrusive, value and pointer lists.
- wclistit.h** This header file defines the `iterator` classes that correspond to the list containers.

## 6 Header Files



**wcqueue.h** This header file defines the `queue` class. Entries in a queue class are accessed first in, first out.

**wcstack.h** This header file defines the `stack` class. Entries in a stack class are accessed last in, first out.



---

## 2 Common Types

The set of classes that make up the C++ class library use several common typedefs and macros. They are declared in `<iostream.h>` and `<fstream.h>`.

```
typedef long streampos;  
typedef long streamoff;  
typedef int filedesc;  
#define __NOT_EOF 0  
#define EOF -1
```

The `streampos` type represents an absolute position within the file. For Open Watcom C++, the file position can be represented by an integral type. For some file systems, or at a lower level within the file system, the stream position might be represented by an aggregate (structure) containing information such as cylinder, track, sector and offset.

The `streamoff` type represents a relative position within the file. The offset can always be represented as a signed integer quantity since it is a number of characters before or after an absolute position within the file.

The `filedesc` type represents the type of a C library file handle. It is used in places where the I/O stream library takes a C library file handle as an argument.

The `__NOT_EOF` macro is defined for cases where a function needs to return something other than `EOF` to indicate success.

The `EOF` macro is defined to be identical to the value provided by the `<stdio.h>` header file.



---

## 3 Predefined Objects

Most programs interact in some manner with the keyboard and screen. The C programming language provides three values, `stdin`, `stdout` and `stderr`, that are used for communicating with these "standard" devices, which are opened before the user program starts execution at `main()`. These three values are `FILE` pointers and can be used in virtually any file operation supported by the C library.

In a similar manner, C++ provides seven objects for communicating with the same "standard" devices. C++ provides the three C `FILE` pointers `stdin`, `stdout` and `stderr`, but they cannot be used with the extractors and inserters provided as part of the C++ library. C++ provides four new objects, called `cin`, `cout`, `cerr` and `clog`, which correspond to `stdin`, `stdout`, `stderr` and buffered `stderr`.

### 3.1 `cin`

`cin` is an `istream` object which is connected to "standard input" (usually the keyboard) prior to program execution. Values extracted using the `istream` operator `>>` class extractor operators are read from standard input and interpreted according to the type of the object being extracted.

Extractions from standard input via `cin` skip whitespace characters by default because the `ios::skipws` bit is on. The default behavior can be changed with the `ios::setf` public member function or with the `setiosflags` manipulator.

### 3.2 `cout`

`cout` is an `ostream` object which is connected to "standard output" (usually the screen) prior to program execution. Values inserted using the `ostream` operator `<<` class inserter operators are converted to characters and written to standard output according to the type of the object being inserted.

Insertions to standard output via `cout` are buffered by default because the `ios::unitbuf` bit is not on. The default behavior can be changed with the `ios::setf` public member function or with the `setiosflags` manipulator.

### 3.3 *cerr*

`cerr` is an `ostream` object which is connected to "standard error" (the screen) prior to program execution. Values inserted using the `ostream` operator `<<` class inserter operators are converted to characters and written to standard error according to the type of the object being inserted.

Insertions to standard error via `cerr` are not buffered by default because the `ios::unitbuf` bit is on. The default behavior can be changed with the `ios::setf` public member function or with the `setiosflags` manipulator.

### 3.4 *clog*

`clog` is an `ostream` object which is connected to "standard error" (the screen) prior to program execution. Values inserted using the `ostream` operator `<<` class inserter operators are converted to characters and written to standard error according to the type of the object being inserted.

Insertions to standard error via `clog` are buffered by default because the `ios::unitbuf` bit is not on. The default behavior can be changed with the `ios::setf` public member function or with the `setiosflags` manipulator.

---

# 4 *istream* Input

This chapter describes formatted and unformatted input.

## 4.1 *Formatted Input: Extractors*

The `operator >>` function is used to read formatted values from a stream. It is called an *extractor*. Characters are read and interpreted according to the type of object being extracted.

All `operator >>` functions perform the same basic sequence of operations. First, the input prefix function `ipfx` is called with a parameter of zero, causing leading whitespace characters to be discarded if `ios::skipws` is set in `ios::fmtflags`. If the input prefix function fails and returns zero, the `operator >>` function also fails and returns immediately. If the input prefix function succeeds, characters are read from the stream and interpreted in terms of the type of object being extracted and `ios::fmtflags`. Finally, the input suffix function `isfx` is called.

The `operator >>` functions return a reference to the specified stream so that multiple extractions can be done in one statement.

Errors are indicated via `ios::iostate`. `ios::failbit` is set if the characters read from the stream could not be interpreted for the required type. `ios::badbit` is set if the extraction of characters from the stream failed in such a way as to make subsequent extractions impossible. `ios::eofbit` is set if the stream was located at the end when the extraction was attempted.

## 4.2 *Unformatted Input*

The unformatted input functions are used to read characters from the stream without interpretation.

Like the extractors, the unformatted input functions follow a pattern. First, they call `ipfx`, the input prefix function, with a parameter of one, causing no leading whitespace characters to be discarded. If the input prefix function fails and returns zero, the unformatted input function also fails and returns immediately. If the input prefix function succeeds, characters

are read from the stream without interpretation. Finally, `isfx`, the input suffix function, is called.

Errors are indicated via the `iostate` bits. `ios::failbit` is set if the extraction of characters from the stream failed. `ios::eofbit` is set if the stream was located at the end of input when the operation was attempted.



---

# 5 *ostream* Output

This chapter describes formatted and unformatted output.

## 5.1 *Formatted Output: Inserters*

The `operator <<` function is used to write formatted values to a stream. It is called an *inserter*. Values are formatted and written according to the type of object being inserted and `ios::fmtflags`.

All `operator <<` functions perform the same basic sequence of operations. First, the output prefix function `opfx` is called. If it fails and returns zero, the `operator <<` function also fails and returns immediately. If the output prefix function succeeds, the object is formatted according to its type and `ios::fmtflags`. The formatted sequence of characters is then written to the specified stream. Finally, the output suffix function `osfx` is called.

The `operator <<` functions return a reference to the specified stream so that multiple insertions can be done in one statement.

For details on the interpretation of `ios::fmtflags`, see the `ios::fmtflags` section of the Library Functions and Types Chapter.

Errors are indicated via `ios::iostate`. `ios::failbit` is set if the `operator <<` function fails while writing the characters to the stream.

## 5.2 *Unformatted Output*

The unformatted output functions are used to write characters to the stream without conversion.

Like the inserters, the unformatted output functions follow a pattern. First, they call the output prefix function `opfx` and fail if it fails. Then the characters are written without conversion. Finally, the output suffix function `osfx` is called.

Errors are indicated via `ios::iostate`. `ios::failbit` is set if the function fails while writing the characters to the stream.

---

## 6 Library Functions and Types

Each of the classes and functions in the Class Library is described in this chapter. Each description consists of a number of subsections:

**Declared:** This optional subsection specifies which header file contains the declaration for a class. It is only found in sections describing class declarations.

**Derived From:**

This optional subsection shows the inheritance for a class. It is only found in sections describing class declarations.

**Derived By:** This optional subsection shows which classes inherit from this class. It is only found in sections describing class declarations.

**Synopsis:** This subsection gives the name of the header file that contains the declaration of the function. This header file must be included in order to reference the function.

For class member functions, the protection associated with the function is indicated via the presence of one of the `private`, `protected`, or `public` keywords.

The full function prototype is specified. Virtual class member functions are indicated via the presence of the `virtual` keyword in the function prototype.

**Semantics:** This subsection is a description of the function.

**Derived Implementation Protocol:**

This optional subsection is present for virtual member functions. It describes how derived implementations of the virtual member function should behave.

**Default Implementation:**

This optional subsection is present for virtual member functions. It describes how the default implementation provided with the base class definition behaves.

**Results:** This optional subsection describes the function's return value, if any, and the impact of a member function on its object's state.

**See Also:** This optional subsection provides a list of related functions or classes.



---

# 7 *Complex Class*

This class is used for the storage and manipulation of complex numbers, which are often represented by *real* and *imaginary* components (Cartesian coordinates), or by *magnitude* and *angle* (polar coordinates). Each object stores exactly one complex number. An object may be used in expressions in the same manner as floating-point values.

The class documented here is the Open Watcom legacy complex class. It is not the `std::complex` class template specified by Standard C++.

**Declared:** `complex.h`

The `Complex` class is used for the storage and manipulation of complex numbers, which are often represented by *real* and *imaginary* components (Cartesian coordinates), or by *magnitude* and *angle* (polar coordinates). Each `Complex` object stores exactly one complex number. A `Complex` object may be used in expressions in the same manner as floating-point values.

### Public Member Functions

The following constructors and destructors are declared:

```
Complex();  
Complex( Complex const & );  
Complex( double, double = 0.0 );  
~Complex();
```

The following arithmetic member functions are declared:

```
Complex &operator =( Complex const & );  
Complex &operator =( double );  
Complex &operator +=( Complex const & );  
Complex &operator +=( double );  
Complex &operator -=( Complex const & );  
Complex &operator -=( double );  
Complex &operator *=( Complex const & );  
Complex &operator *=( double );  
Complex &operator /=( Complex const & );  
Complex &operator /=( double );  
Complex operator +( ) const;  
Complex operator -( ) const;  
double imag() const;  
double real() const;
```

### Friend Functions

The following I/O Stream inserter and extractor friend functions are declared:

```
friend istream &operator >>( istream &, Complex & );  
friend ostream &operator <<( ostream &, Complex const & );
```

### Related Operators

The following operators are declared:

```
Complex operator +( Complex const &, Complex const & );
```

```

Complex operator +( Complex const &, double );
Complex operator +( double          , Complex const & );
Complex operator -( Complex const &, Complex const & );
Complex operator -( Complex const &, double );
Complex operator -( double          , Complex const & );
Complex operator *( Complex const &, Complex const & );
Complex operator *( Complex const &, double );
Complex operator *( double          , Complex const & );
Complex operator /( Complex const &, Complex const & );
Complex operator /( Complex const &, double );
Complex operator /( double          , Complex const & );
int operator ==( Complex const &, Complex const & );
int operator ==( Complex const &, double );
int operator ==( double          , Complex const & );
int operator !=( Complex const &, Complex const & );
int operator !=( Complex const &, double );
int operator !=( double          , Complex const & );

```

### Related Functions

The following related functions are declared:

```

double abs ( Complex const & );
Complex acos ( Complex const & );
Complex acosh( Complex const & );
double arg ( Complex const & );
Complex asin ( Complex const & );
Complex asinh( Complex const & );
Complex atan ( Complex const & );
Complex atanh( Complex const & );
Complex conj ( Complex const & );
Complex cos ( Complex const & );
Complex cosh ( Complex const & );
Complex exp ( Complex const & );
double imag ( Complex const & );
Complex log ( Complex const & );
Complex log10( Complex const & );
double norm ( Complex const & );
Complex polar( double          , double = 0 );
Complex pow ( Complex const &, Complex const & );
Complex pow ( Complex const &, double );
Complex pow ( double          , Complex const & );
Complex pow ( Complex const &, int );
double real ( Complex const & );
Complex sin ( Complex const & );
Complex sinh ( Complex const & );
Complex sqrt ( Complex const & );

```

## ***Complex***

---

```
Complex tan ( Complex const & );  
Complex tanh ( Complex const & );
```



**Synopsis:**

```
#include <complex.h>
double abs( Complex const &num );
```

**Semantics:** The `abs` function computes the magnitude of *num*, which is equivalent to the length (magnitude) of the vector when the *num* is represented in polar coordinates.

**Results:** The `abs` function returns the magnitude of *num*.

**See Also:** `arg`, `norm`, `polar`

## ***Complex acos()***

---

**Synopsis:** `#include <complex.h>`  
`Complex acos( Complex const &num );`

**Semantics:** The `acos` function computes the arccosine of *num*.

**Results:** The `acos` function returns the arccosine of *num*.

**See Also:** `asin`, `atan`, `cos`

**Synopsis:**

```
#include <complex.h>
Complex acosh( Complex const &num );
```

**Semantics:** The *acosh* function computes the inverse hyperbolic cosine of *num*.

**Results:** The *acosh* function returns the inverse hyperbolic cosine of *num*.

**See Also:** *asinh*, *atanh*, *cosh*

## Complex *arg()*

---

**Synopsis:**

```
#include <complex.h>
double arg( Complex const &num );
```

**Semantics:** The `arg` function computes the angle of the vector when the *num* is represented in polar coordinates. The angle has the same sign as the real component of the *num*. It is positive in the 1st and 2nd quadrants, and negative in the 3rd and 4th quadrants.

**Results:** The `arg` function returns the angle of the vector when the *num* is represented in polar coordinates.

**See Also:** `abs`, `norm`, `polar`

**Synopsis:**

```
#include <complex.h>
Complex asin( Complex const &num );
```

**Semantics:** The `asin` function computes the arcsine of *num*.

**Results:** The `asin` function returns the arcsine of *num*.

**See Also:** `acos`, `atan`, `sin`

## ***Complex asinh()***

---

**Synopsis:** `#include <complex.h>`  
`Complex asinh( Complex const &num );`

**Semantics:** The `asinh` function computes the inverse hyperbolic sine of *num*.

**Results:** The `asinh` function returns the inverse hyperbolic sine of *num*.

**See Also:** `acosh`, `atanh`, `sinh`

**Synopsis:**

```
#include <complex.h>
Complex atan( Complex const &num );
```

**Semantics:** The atan function computes the arctangent of *num*.

**Results:** The atan function returns the arctangent of *num*.

**See Also:** acos, asin, tan

## ***Complex atanh()***

---

**Synopsis:** `#include <complex.h>`  
`Complex atanh( Complex const &num );`

**Semantics:** The `atanh` function computes the inverse hyperbolic tangent of *num*.

**Results:** The `atanh` function returns the inverse hyperbolic tangent of *num*.

**See Also:** `acosh`, `asinh`, `tanh`



**Synopsis:**

```
#include <complex.h>
public:
Complex::Complex();
```

**Semantics:** This form of the public `Complex` constructor creates a default `Complex` object with value zero for both the real and imaginary components.

**Results:** This form of the public `Complex` constructor produces a default `Complex` object.

**See Also:** `~Complex`, `real`, `imag`

## ***Complex::Complex()***

---

**Synopsis:**

```
#include <complex.h>
public:
Complex::Complex( Complex const &num );
```

**Semantics:** This form of the public `Complex` constructor creates a `Complex` object with the same value as *num*.

**Results:** This form of the public `Complex` constructor produces a `Complex` object.

**See Also:** `~Complex`, `real`, `imag`

**Synopsis:**

```
#include <complex.h>
public:
Complex::Complex( double real, double imag = 0.0 );
```

**Semantics:** This form of the public `Complex` constructor creates a `Complex` object with the real component set to *real* and the imaginary component set to *imag*. If no imaginary component is specified, *imag* takes the default value of zero.

**Results:** This form of the public `Complex` constructor produces a `Complex` object.

**See Also:** `~Complex`, `real`, `imag`

## ***Complex::~~Complex()***

---

**Synopsis:**

```
#include <complex.h>
public:
Complex::~~Complex();
```

**Semantics:** The public `~Complex` destructor destroys the `Complex` object. The call to the public `~Complex` destructor is inserted implicitly by the compiler at the point where the `Complex` object goes out of scope.

**Results:** The `Complex` object is destroyed.

**See Also:** `Complex`

**Synopsis:**

```
#include <complex.h>
Complex conj( Complex const &num );
```

**Semantics:** The `conj` function computes the conjugate of *num*. The conjugate consists of the unchanged real component, and the negative of the imaginary component.

**Results:** The `conj` function returns the conjugate of *num*.

## ***Complex cos()***

---

**Synopsis:** `#include <complex.h>`  
`Complex cos( Complex const &num );`

**Semantics:** The `cos` function computes the cosine of *num*.

**Results:** The `cos` function returns the cosine of *num*.

**See Also:** `acos`, `sin`, `tan`

**Synopsis:** `#include <complex.h>`  
`Complex cosh( Complex const &num );`

**Semantics:** The `cosh` function computes the hyperbolic cosine of *num*.

**Results:** The `cosh` function returns the hyperbolic cosine of *num*.

**See Also:** `acosh`, `sinh`, `tanh`

## ***Complex exp()***

---

**Synopsis:** `#include <complex.h>`  
`Complex exp( Complex const &num );`

**Semantics:** The `exp` function computes the value of **e** raised to the power *num*.

**Results:** The `exp` function returns the value of **e** raised to the power *num*.

**See Also:** `log`, `log10`, `pow`, `sqrt`



**Synopsis:**

```
#include <complex.h>
public:
double Complex::imag();
```

**Semantics:** The `imag` public member function extracts the imaginary component of the `Complex` object.

**Results:** The `imag` public member function returns the imaginary component of the `Complex` object.

**See Also:** `imag`, `real`  
`Complex::real`

## ***Complex imag()***

---

**Synopsis:** `#include <complex.h>`  
`double imag( Complex const &num );`

**Semantics:** The `imag` function extracts the imaginary component of *num*.

**Results:** The `imag` function returns the imaginary component of *num*.

**See Also:** `real`  
`Complex::imag`, `real`

**Synopsis:**

```
#include <complex.h>
Complex log( Complex const &num );
```

**Semantics:** The `log` function computes the natural, or base **e**, logarithm of *num*.

**Results:** The `log` function returns the natural, or base **e**, logarithm of *num*.

**See Also:** `exp`, `log10`, `pow`, `sqrt`

## ***Complex log10()***

---

**Synopsis:** `#include <complex.h>`  
`Complex log10( Complex const &num );`

**Semantics:** The `log10` function computes the base 10 logarithm of *num*.

**Results:** The `log10` function returns the base 10 logarithm of *num*.

**See Also:** `exp`, `log`, `pow`, `sqrt`

**Synopsis:**

```
#include <complex.h>
double norm( Complex const &num );
```

**Semantics:** The norm function computes the square of the magnitude of *num*, which is equivalent to the square of the length (magnitude) of the vector when *num* is represented in polar coordinates.

**Results:** The norm function returns the square of the magnitude of *num*.

**See Also:** `arg`, `polar`

## Complex operator !=()

---

**Synopsis:**

```
#include <complex.h>
int operator !=( Complex const &num1, Complex const &num2 );
int operator !=( Complex const &num1, double num2 );
int operator !=( double num1, Complex const &num2 );
```

**Semantics:** The operator `!=` function compares *num1* and *num2* for inequality. At least one of the parameters must be a `Complex` object for this function to be called.

Two `Complex` objects are not equal if either of their corresponding real or imaginary components are not equal.

If the operator `!=` function is used with a `Complex` object and an object of any other built-in numeric type, the non-`Complex` object is converted to a `double` and the second or third form of the operator `!=` function is used.

**Results:** The operator `!=` function returns a non-zero value if *num1* is not equal to *num2*, otherwise zero is returned.

**See Also:** `operator ==`

**Synopsis:**

```
#include <complex.h>
Complex operator *( Complex const &num1,
Complex const &num2 );
Complex operator *( Complex const &num1,
double num2 );
Complex operator *( double num1,
Complex const &num2 );
```

**Semantics:** The operator `*` function is used to multiply *num1* by *num2* yielding a `Complex` object.

The first operator `*` function multiplies two `Complex` objects.

The second operator `*` function multiplies a `Complex` object and a floating-point value. In effect, the real and imaginary components of the `Complex` object are multiplied by the floating-point value.

The third operator `*` function multiplies a floating-point value and a `Complex` object. In effect, the real and imaginary components of the `Complex` object are multiplied by the floating-point value.

If the operator `*` function is used with a `Complex` object and an object of any other built-in numeric type, the non-`Complex` object is converted to a `double` and the second or third form of the operator `*` function is used.

**Results:** The operator `*` function returns a `Complex` object that is the product of *num1* and *num2*.

**See Also:** `operator +`, `operator -`, `operator /`  
`Complex::operator *=`

## ***Complex::operator \*=( )***

---

**Synopsis:** `#include <complex.h>`  
`public:`  
`Complex &Complex::operator *=( Complex const &num );`  
`Complex &Complex::operator *=( double num );`

**Semantics:** The operator `*` public member function is used to multiply the *num* argument into the `Complex` object.

The first form of the operator `*` public member function multiplies the `Complex` object by the `Complex` parameter.

The second form of the operator `*` public member function multiplies the real and imaginary components of the `Complex` object by *num*.

A call to the operator `*` public member function where *num* is any of the other built-in numeric types, causes *num* to be promoted to `double` and the second form of the operator `*` public member function to be used.

**Results:** The operator `*` public member function returns a reference to the target of the assignment.

**See Also:** `operator *`  
`Complex::operator +=`, `operator -=`, `operator /=`, `operator =`



**Synopsis:**

```
#include <complex.h>
public:
Complex Complex::operator +();
```

**Semantics:** The unary `operator +` public member function is provided for completeness. It performs no operation on the `Complex` object.

**Results:** The unary `operator +` public member function returns a `Complex` object with the same value as the original `Complex` object.

**See Also:** `operator +`  
`Complex::operator +=`, `operator -`

## Complex operator +()

---

**Synopsis:**

```
#include <complex.h>
Complex operator +( Complex const &num1,
Complex const &num2 );
Complex operator +( Complex const &num1,
double num2 );
Complex operator +( double num1,
Complex const &num2 );
```

**Semantics:** The operator `+` function is used to add *num1* to *num2* yielding a `Complex` object.

The first operator `+` function adds two `Complex` objects.

The second operator `+` function adds a `Complex` object and a floating-point value. In effect, the floating-point value is added to the real component of the `Complex` object.

The third operator `+` function adds a floating-point value and a `Complex` object. In effect, the floating-point value is added to the real component of the `Complex` object.

If the operator `+` function is used with a `Complex` object and an object of any other built-in numeric type, the non-`Complex` object is converted to a `double` and the second or third form of the operator `+` function is used.

**Results:** The operator `+` function returns a `Complex` object that is the sum of *num1* and *num2*.

**See Also:** `operator *`, `operator -`, `operator /`  
`Complex::operator +`, `operator +=`

**Synopsis:**

```
#include <complex.h>
public:
Complex &Complex::operator +=( Complex const &num );
Complex &Complex::operator +=( double num );
```

**Semantics:** The `operator +=` public member function is used to add *num* to the value of the `Complex` object. The second form of the `operator +=` public member function adds *num* to the real component of the `Complex` object.

A call to the `operator +=` public member function where *num* is any of the other built-in numeric types, causes *num* to be promoted to `double` and the second form of the `operator +=` public member function to be used.

**Results:** The `operator +=` public member function returns a reference to the target of the assignment.

**See Also:** `operator +`,  
`Complex::operator * =`, `operator +`, `operator / =`, `operator - =`,  
`operator =`

## ***Complex::operator -()***

---

**Synopsis:** `#include <complex.h>`  
`public:`  
`Complex Complex::operator -();`

**Semantics:** The unary operator `-` public member function yields a `Complex` object with the real and imaginary components having the same magnitude as those of the original object, but with opposite sign.

**Results:** The unary operator `-` public member function returns a `Complex` object with the same magnitude as the original `Complex` object and with opposite sign.

**See Also:** `operator -`  
`Complex::operator +`, `operator -=`

**Synopsis:**

```
#include <complex.h>
Complex operator -( Complex const &num1,
Complex const &num2 );
Complex operator -( Complex const &num1,
double num2 );
Complex operator -( double num1,
Complex const &num2 );
```

**Semantics:** The operator `-` function is used to subtract *num2* from *num1* yielding a `Complex` object.

The first operator `-` function computes the difference between two `Complex` objects.

The second operator `-` function computes the difference between a `Complex` object and a floating-point value. In effect, the floating-point value is subtracted from the real component of the `Complex` object.

The third operator `-` function computes the difference between a floating-point value and a `Complex` object. In effect, the real component of the result is *num1* minus the real component of *num2*, and the imaginary component of the result is the negative of the imaginary component of *num2*.

If the operator `-` function is used with a `Complex` object and an object of any other built-in numeric type, the non-`Complex` object is converted to a `double` and the second or third form of the operator `-` function is used.

**Results:** The operator `-` function returns a `Complex` object that is the difference between *num1* and *num2*.

**See Also:** `operator *`, `operator +`, `operator /`  
`Complex::operator -`, `operator -=`

## ***Complex::operator -=()***

---

**Synopsis:**

```
#include <complex.h>
public:
Complex &Complex::operator -=( Complex const &num );
Complex &Complex::operator -=( double num );
```

**Semantics:** The operator `-=` public member function is used to subtract *num* from the value of the `Complex` object. The second form of the operator `-=` public member function subtracts *num* from the real component of the `*obj`.

A call to the operator `-=` public member function where *num* is any of the other built-in numeric types, causes *num* to be promoted to `double` and the second form of the operator `-=` public member function to be used.

**Results:** The operator `-=` public member function returns a reference to the target of the assignment.

**See Also:** `operator -`  
`Complex::operator * =`, `operator +=`, `operator -`, `operator /=`,  
`operator =`

**Synopsis:**

```
#include <complex.h>
Complex operator /( Complex const &num1,
Complex const &num2 );
Complex operator /( Complex const &num1,
double num2 );
Complex operator /( double num1,
Complex const &num2 );
```

**Semantics:** The operator `/` function is used to divide *num1* by *num2* yielding a `Complex` object.

The first operator `/` function divides two `Complex` objects.

The second operator `/` function divides a `Complex` object by a floating-point value. In effect, the real and imaginary components of the complex number are divided by the floating-point value.

The third operator `/` function divides a floating-point value by a `Complex` object. Conceptually, the floating-point value is converted to a `Complex` object and then the division is done.

If the operator `/` function is used with a `Complex` object and an object of any other built-in numeric type, the non-`Complex` object is converted to a `double` and the second or third form of the operator `/` function is used.

**Results:** The operator `/` function returns a `Complex` object that is the quotient of *num1* divided by *num2*.

**See Also:** `operator *`, `operator +`, `operator -`  
`Complex::operator /=`

## ***Complex::operator /=()***

---

**Synopsis:**

```
#include <complex.h>
public:
Complex &Complex::operator /=( Complex const &num );
Complex &Complex::operator /=( double num );
```

**Semantics:** The `operator /=` public member function is used to divide the `Complex` object by *num*. The second form of the `operator /=` public member function divides the real and imaginary components of the `Complex` object by *num*.

A call to the `operator /=` public member function where *num* is any of the other built-in numeric types, causes *num* to be promoted to `double` and the second form of the `operator /=` public member function to be used.

**Results:** The `operator /=` public member function returns a reference to the target of the assignment.

**See Also:** `operator /`  
`Complex::operator *=`, `operator +=`, `operator -=`, `operator =`



**Synopsis:**

```
#include <complex.h>
friend ostream &operator <<( ostream &strm, Complex &num );
```

**Semantics:** The operator << function is used to write `Complex` objects to an I/O stream. The `Complex` object is always written in the form:

```
( real, imag )
```

The real and imaginary components are written using the normal rules for formatting floating-point numbers. Any formatting options specified prior to inserting the *num* apply to both the real and imaginary components. If the real and imaginary components are to be inserted using different formats, the `real` and `imag` member functions should be used to insert each component separately.

**Results:** The operator << function returns a reference to the *strm* object.

**See Also:** `istream`

## ***Complex::operator =()***

---

**Synopsis:**

```
#include <complex.h>
public:
Complex &Complex::operator =( Complex const &num );
Complex &Complex::operator =( double num );
```

**Semantics:** The `operator =` public member function is used to set the value of the `Complex` object to `num`. The first assignment operator copies the value of `num` into the `Complex` object.

The second assignment operator sets the real component of the `Complex` object to `num` and the imaginary component to zero.

A call to the `operator =` public member function where `num` is any of the other built-in numeric types, causes `num` to be promoted to `double` and the second form of the `operator =` public member function to be used.

**Results:** The `operator =` public member function returns a reference to the target of the assignment.

**See Also:** `Complex::operator *=`, `operator +=`, `operator -=`, `operator /=`

**Synopsis:**

```
#include <complex.h>
int operator ==( Complex const &num1, Complex const &num2 );
int operator ==( Complex const &num1, double num2 );
int operator ==( double num1, Complex const &num2 );
```

**Semantics:** The `operator ==` function compares *num1* and *num2* for equality. At least one of the arguments must be a `Complex` object for this function to be called.

Two `Complex` objects are equal if their corresponding real and imaginary components are equal.

If the `operator ==` function is used with a `Complex` object and an object of any other built-in numeric type, the non-`Complex` object is converted to a `double` and the second or third form of the `operator ==` function is used.

**Results:** The `operator ==` function returns a non-zero value if *num1* is equal to *num2*, otherwise zero is returned.

**See Also:** `operator !=`

## Complex operator >>()

---

**Synopsis:**

```
#include <complex.h>
friend istream &operator >>( istream &strm, Complex &num );
```

**Semantics:** The operator `>>` function is used to read a `Complex` object from an I/O stream. A valid complex value is of one of the following forms:

```
( real, imag)
real, imag
(real)
```

If the imaginary portion is omitted, zero is assumed.

While reading a `Complex` object, whitespace is ignored before and between the various components of the number if the `ios::skipws` bit is set in `ios::fmtflags`.

**Results:** The operator `>>` function returns a reference to `strm`. `num` contains the value read from `strm` on success, otherwise it is unchanged.

**See Also:** `istream`

**Synopsis:**

```
#include <complex.h>
Complex polar( double mag, double angle = 0.0 );
```

**Semantics:** The `polar` function converts *mag* and *angle* (polar coordinates) into a complex number. The *angle* is optional and defaults to zero if it is unspecified.

**Results:** The `polar` function returns a `Complex` object that is *mag* and *angle* interpreted as polar coordinates.

**See Also:** `abs`, `arg`, `norm`

## Complex pow()

---

**Synopsis:**

```
#include <complex.h>
Complex pow( Complex const &num, Complex const &exp );
Complex pow( Complex const &num, double exp );
Complex pow( double num, Complex const &exp );
Complex pow( Complex const &num, int exp );
```

**Semantics:** The `pow` function computes *num* raised to the power *exp*. The various forms are provided to minimize the amount of floating-point calculation performed.

**Results:** The `pow` function returns a Complex object that is *num* raised to the power a Complex object that is *exp*.

**See Also:** `exp`, `log`, `log10`, `sqrt`

**Synopsis:**

```
#include <complex.h>
public:
double Complex::real();
```

**Semantics:** The `real` public member function extracts the real component of the `Complex` object.

**Results:** The `real` public member function returns the real component of the `Complex` object.

**See Also:** `imag`, `real`  
`Complex::imag`

## ***Complex real()***

---

**Synopsis:** `#include <complex.h>`  
`double real( Complex const &num );`

**Semantics:** The `real` function extracts the real component of *num*.

**Results:** The `real` function returns the real component of *num*.

**See Also:** `imag`  
`Complex::imag`, `real`



**Synopsis:**

```
#include <complex.h>
Complex sin( Complex const &num );
```

**Semantics:** The `sin` function computes the sine of *num*.

**Results:** The `sin` function returns the sine of *num*.

**See Also:** `asin`, `cos`, `tan`

## ***Complex sinh()***

---

**Synopsis:** `#include <complex.h>`  
`Complex sinh( Complex const &num );`

**Semantics:** The `sinh` function computes the hyperbolic sine of *num*.

**Results:** The `sinh` function returns the hyperbolic sine of *num*.

**See Also:** `asinh`, `cosh`, `tanh`

**Synopsis:**

```
#include <complex.h>
Complex sqrt( Complex const &num );
```

**Semantics:** The `sqrt` function computes the square root of *num*.

**Results:** The `sqrt` function returns the square root of *num*.

**See Also:** `exp`, `log`, `log10`, `pow`

## ***Complex tan()***

---

**Synopsis:** `#include <complex.h>`  
`Complex tan( Complex const &num );`

**Semantics:** The `tan` function computes the tangent of *num*.

**Results:** The `tan` function returns the tangent of *num*.

**See Also:** `atan`, `cos`, `sin`

**Synopsis:**

```
#include <complex.h>
Complex tanh( Complex const &num );
```

**Semantics:** The `tanh` function computes the hyperbolic tangent of *num*.

**Results:** The `tanh` function returns the hyperbolic tangent of *num*.

**See Also:** `atanh`, `cosh`, `sinh`



---

# ***8 Container Exception Classes***

This chapter describes exception handling for the container classes.

**Declared:** `wcexcept.h`

The `WCExcept` class provides the exception handling for the container classes. If you have compiled your code with exception handling enabled, the C++ exception processing can be used to catch errors. Your source file must be compiled with the exception handling compile switch for C++ exception processing to occur. The container classes will attempt to set the container object into a reasonable state if there is an error and exception handling is not enabled, or if the trap for the specific error has not been enabled by your program.

By default, no exception traps are enabled and no exceptions will be thrown. Exception traps are enabled by setting the exception state with the `exceptions` member function.

The `wcexcept.h` header file is included by the header files for each of the container classes. There is normally no need to explicitly include the `wcexcept.h` header file, but no errors will result if it is included. This class is inherited as a base class for each of the containers. You do not need to derive from it directly.

The `WCListExcept` class (formally used by the list container classes) has been replaced by the `WCExcept` class. A typedef of the `WCListExcept` class to the `WCExcept` class and the `wclist_state` type to the `wc_state` type provide backward compatibility with previous versions of the list containers.

### Public Enumerations

The following enumeration typedefs are declared in the public interface:

```
typedef int wc_state;
```

### Public Member Functions

The following public member functions are declared:

```
WCExcept();  
virtual ~WCExcept();  
wc_state exceptions() const;  
wc_state exceptions( wc_state );
```



**Synopsis:**

```
#include <wceexcept.h>
public:
WCEexcept();
```

**Semantics:** This form of the public `WCEexcept` constructor creates an `WCEexcept` object.

The public `WCEexcept` constructor is used implicitly by the compiler when it generates a constructor for a derived class. It is automatically used by the list container classes, and should not be required in any user derived classes.

**Results:** The public `WCEexcept` constructor produces an initialized `WCEexcept` object with no exception traps enabled.

**See Also:** `~WCEexcept`

## ***WCEexcept::~WCEexcept()***

---

**Synopsis:**

```
#include <wceexcept.h>
public:
virtual ~WCEexcept();
```

**Semantics:** The public `~WCEexcept` destructor does not do anything explicit. The call to the public `~WCEexcept` destructor is inserted implicitly by the compiler at the point where the object derived from `WCEexcept` goes out of scope.

**Results:** The object derived from `WCEexcept` is destroyed.

**See Also:** `WCEexcept`

**Synopsis:**

```
#include <wexcept.h>
public:
wc_state exceptions() const;
wc_state exceptions( wc_state set_flags );
```

**Semantics:** The `exceptions` public member function queries and/or sets the bits that control which exceptions are enabled for the list class. Each bit corresponds to an exception, and is set if the exception is enabled. The first form of the `exceptions` public member function returns the current settings of the exception bits. The second form of the function sets the exception bits to those specified by *set\_flags*.

**Results:** The current exception bits are returned. If a new set of bits are being set, the returned value is the old set of exception bits.

**Synopsis:**

```
#include <wexcept.h>
public:
enum wcstate {
all_fine = 0x0000, // - no errors
check_none = all_fine, // - throw no exceptions
not_empty = 0x0001, // - container not empty
index_range = 0x0002, // - index is out of range
empty_container= 0x0004, // - empty container error
out_of_memory = 0x0008, // - allocation failed
resize_required= 0x0010, // - request needs resize
not_unique = 0x0020, // - adding duplicate
zero_buckets = 0x0040, // - resizing hash to zero
// value to use to check for all errors
check_all = (not_empty|index_range|empty_container
|out_of_memory|resize_required
|not_unique|zero_buckets)
};
typedef int wc_state;
```

**Semantics:** The type `WCExcept::wcstate` is a set of bits representing the current state of the container object. The `WCExcept::wc_state` member typedef represents the same set of bits, but uses an `int` to represent the values, thereby avoiding problems made possible by the compiler's ability to use smaller types for enumerations. All uses of these bits should use the `WCExcept::wc_state` member typedef.

The bit values defined by the `WCExcept::wc_state` member typedef can be read and set by the `exceptions` member function, which is also used to control exception handling.

The `WCExcept::not_empty` bit setting traps the destruction of a container when the container has at one or more entries. If this error is not trapped, memory may not be properly released back to the system.

The `WCExcept::index_range` state setting traps an attempt to access a container item by an index value that is either not positive or is larger than the index of the last item in the container.

The `WCExcept::empty_container` bit setting traps an attempt to perform an invalid operation on a container with no entries.

The `WCExcept::out_of_memory` bit setting traps any container class allocation failures. If this exception is not enabled, the operation in which the allocation failed will return a `FALSE` (zero) value. Container class copy constructors and assignment operators can also throw this exception, and if not enabled incomplete copies may result.

The `WCEexcept::resize_required` bit setting traps any vector operations which cannot be performed unless the vector is resized to a larger size. If this exception is not enabled, the vector class will attempt an appropriate resize when necessary for an operation.

The `WCEexcept::not_unique` bit setting traps an attempt to add a duplicate value to a set container, or a duplicate key to a dictionary container. The duplicate value is not added to the container object regardless of the exception trap state.

The `WCEexcept::zero_buckets` bit setting traps an attempt to resize of hash container to have zero buckets. No resize is performed whether or not the exception is enabled.

**Declared:** `wcexcept.h`

The `WCIterExcept` class provides the exception handling for the container iterators. If you have compiled your code with exception handling enabled, the C++ exception processing can be used to catch errors. Your source file must be compiled with the exception handling compile switch for C++ exception processing to occur. The iterators will attempt to set the class into a reasonable state if there is an error and exception handling is not enabled, or if the trap for the specific error has not been enabled by your program.

By default, no exception traps are enabled and no exceptions will be thrown. Exception traps are enabled by setting the exception state with the `exceptions` member function.

The `wcexcept.h` header file is included by the header files for each of the iterator classes. There is normally no need to explicitly include the `wcexcept.h` header file, but no errors will result if it is included. This class is inherited as part of the base construction for each of the iterators. You do not need to derive from it directly.

### Public Enumerations

The following enumeration typedefs are declared in the public interface:

```
typedef int wciter_state;
```

### Public Member Functions

The following public member functions are declared:

```
WCIterExcept();  
virtual ~WCIterExcept();  
wciter_state exceptions() const;  
wciter_state exceptions( wciter_state );
```

**Synopsis:**

```
#include <wexcept.h>
public:
WCIterExcept();
```

**Semantics:** This form of the public `WCIterExcept` constructor creates an `WCIterExcept` object.

The public `WCIterExcept` constructor is used implicitly by the compiler when it generates a constructor for a derived class.

**Results:** The public `WCIterExcept` constructor produces an initialized `WCIterExcept` object with no exception traps enabled.

**See Also:** `~WCIterExcept`

## ***WCIterExcept::~WCIterExcept()***

---

**Synopsis:**

```
#include <wexcept.h>
public:
virtual ~WCIterExcept();
```

**Semantics:** The public `~WCIterExcept` destructor does not do anything explicit. The call to the public `~WCIterExcept` destructor is inserted implicitly by the compiler at the point where the object derived from `WCIterExcept` goes out of scope.

**Results:** The object derived from `WCIterExcept` is destroyed.

**See Also:** `WCIterExcept`



**Synopsis:**

```
#include <wexcept.h>
public:
wclter_state exceptions() const;
wclter_state exceptions( wclter_state set_flags );
```

**Semantics:** The `exceptions` public member function queries and/or sets the bits that control which exceptions are enabled for the iterator class. Each bit corresponds to an exception, and is set if the exception is enabled. The first form of the `exceptions` public member function returns the current settings of the exception bits. The second form of the function sets the exception bits to those specified by *set\_flags*.

**Results:** The current exception bits are returned. If a new set of bits are being set, the returned value is the old set of exception bits.

**Synopsis:**

```
#include <wexcept.h>
public:
enum wciiterstate {
all_fine = 0x0000, // - no errors
check_none = all_fine, // - disable all exceptions
undef_iter = 0x0001, // - position is undefined
undef_item = 0x0002, // - iterator item is undefined
iter_range = 0x0004, // - advance value is bad
// value to use to check for all errors
check_all= (undef_iter|undef_item|iter_range)
};
typedef int wciiter_state;
```

**Semantics:** The type `WCIterExcept::wciiterstate` is a set of bits representing the current state of the iterator. The `WCIterExcept::wciiter_state` member typedef represents the same set of bits, but uses an `int` to represent the values, thereby avoiding problems made possible by the compiler's ability to use smaller types for enumerations. All uses of these bits should use the `WCIterExcept::wciiter_state` member typedef.

The bit values defined by the `WCIterExcept::wciiter_state` member typedef can be read and set by the member function `exceptions`, which is used to control exception handling.

The `WCIterExcept::undef_iter` bit setting traps the use of the iterator when the position within the container object is undefined. Trying to operate on an iterator with no associated container object, increment an iterator which is after the last element, or decrement an iterator positioned before the first element is an undefined operation.

The `WCIterExcept::undef_item` bit setting traps an attempt to obtain the current element of the iterator when the iterator has no associated container object, or is positioned either before or after the container elements. The `undef_item` exception can be thrown only by the `key` and `value` dictionary iterator member functions, and the `current` member function for non-dictionary iterators.

The `WCIterExcept::iter_range` bit setting traps an attempt to use a iteration count value that would place the iterator more than one element past the end or before the beginning of the container elements. The `iter_range` exception can be thrown only by the operator `+=` and operator `-=` operators.

---

# 9 Container Allocators and Deallocators

## Example

```
#include <iostream.h>
#include <wclist.h>
#include <wclistit.h>
#include <wcskip.h>
#include <wskipit.h>
#include <stdlib.h>

#pragma warning 549 9

const int ElemsPerBlock = 50;

//
// Simple block allocation class. Allocate blocks for ElemsPerBlock
// elements, and use part of the block for each of the next ElemsPerBlock
// allocations, incrementing the number allocated elements. Repeat getting
// more blocks as needed.
//
// Store the blocks in an intrusive single linked list.
//
// On a element deallocation, assume we allocated the memory and just
// decrement the count of allocated elements. When the count gets to zero,
// free all allocated blocks
//
// This implementation assumes sizeof( char ) == 1
//

class BlockAlloc {
private:
    // the size of elements (in bytes)
    unsigned elem_size;

    // number of elements allocated
    unsigned num_allocated;

    // free space of this number of elements available in first block
    unsigned num_free_in_block;

    // list of blocks used to store elements (block are chunks of memory,
    // pointed by (char *) pointers.
    WCPtrSList<char> block_list;

    // pointer to the first block in the list
    char *curr_block;

public:
    inline BlockAlloc( unsigned size )
```

```
        : elem_size( size ), num_allocated( 0 )
        , num_free_in_block( 0 ) {}

inline BlockAlloc() {
    block_list.clearAndDestroy();
};

// get memory for an element using block allocation
void *allocator( size_t elem_size );

// free memory for an element using block allocation and deallocation
void deallocator( void *old_ptr, size_t elem_size );
};

void *BlockAlloc::allocator( size_t size ) {
    // need a new block to perform allocation
    if( num_free_in_block == 0 ) {
        // allocate memory for ElemsPerBlock elements
        curr_block = new char[ size * ElemsPerBlock ];
        if( curr_block == 0 ) {
            // allocation failed
            return( 0 );
        }
        // add new block to beginning of list
        if( !block_list.insert( curr_block ) ) {
            // allocation of list element failed
            delete( curr_block );
            return( 0 );
        }
        num_free_in_block = ElemsPerBlock;
    }

    // curr block points to a block of memory with some free memory
    num_allocated++;
    num_free_in_block--;
    // return pointer to a free part of the block, starting at the end
    // of the block
    return( curr_block + num_free_in_block * size );
}

void BlockAlloc::deallocator( void *, size_t ) {
    // just decrement the count
    // don't free anything until all elements are deallocated
    num_allocated--;
    if( num_allocated == 0 ) {
        // all the elements allocated BlockAlloc object have now been
        // deallocated, free all the blocks
        block_list.clearAndDestroy();
        num_free_in_block = 0;
    }
}

const unsigned NumTestElems = 200;
```

```
// array with random elements
static unsigned test_elems[ NumTestElems ];

static void fill_test_elems() {
    for( int i = 0; i < NumTestElems; i++ ) {
        test_elems[ i ] = rand();
    }
}

void test_isv_list();
void test_val_list();
void test_val_skip_list();

void main() {
    fill_test_elems();

    test_isv_list();
    test_val_list();
    test_val_skip_list();
}

// An intrusive list class
class isvInt : public WCSLink {
public:
    static BlockAlloc memory_manage;
    int data;

    isvInt( int datum ) : data( datum ) {};

    void *operator new( size_t size ) {
        return( memory_manage.allocator( size ) );
    };

    void operator delete( void *old, size_t size ) {
        memory_manage.deallocator( old, size );
    };
};

// define static member data
BlockAlloc isvInt::memory_manage( sizeof( isvInt ) );

void test_isv_list() {
    WCISvSList<isvInt> list;

    for( int i = 0; i < NumTestElems; i++ ) {
        list.insert( new isvInt( test_elems[ i ] ) );
    }

    WCISvSListIter<isvInt> iter( list );
    while( ++iter ) {
        cout << iter.current()->data << " ";
    }
}
```

```
        cout << "\n\n\n";
        list.clearAndDestroy();
    }

    // WCValsList<int> memory allocator/deallocator support
    static BlockAlloc val_list_manager( WCValsListItemSize( int ) );

    static void *val_list_alloc( size_t size ) {
        return( val_list_manager.allocator( size ) );
    }

    static void val_list_dealloc( void *old, size_t size ) {
        val_list_manager.deallocator( old, size );
    }

    // test WCValsList<int>
    void test_val_list() {
        WCValsList<int> list( &val_list_alloc, &val_list_dealloc );

        for( int i = 0; i < NumTestElems; i++ ) {
            list.insert( test_elems[ i ] );
        }

        WCValsListIter<int> iter( list );
        while( ++iter ) {
            cout << iter.current() << " ";
        }
        cout << "\n\n\n";
        list.clear();
    }

    // skip list allocator deallocators: just use allocator and deallocator
    // functions on skip list elements with one and two pointers
    // (this will handle 94% of the elements)
    const int one_ptr_size = WCValsSkipListItemSize( int, 1 );
    const int two_ptr_size = WCValsSkipListItemSize( int, 2 );

    static BlockAlloc one_ptr_manager( one_ptr_size );
    static BlockAlloc two_ptr_manager( two_ptr_size );

    static void *val_skip_list_alloc( size_t size ) {
        switch( size ) {
            case one_ptr_size:
                return( one_ptr_manager.allocator( size ) );
            case two_ptr_size:
                return( two_ptr_manager.allocator( size ) );
            default:
                return( new char[ size ] );
        }
    }

    static void val_skip_list_dealloc( void *old, size_t size ) {
        switch( size ) {
            case one_ptr_size:
                one_ptr_manager.deallocator( old, size );
                break;
        }
    }
}
```

```
        case two_ptr_size:
            two_ptr_manager.deallocator( old, size );
            break;
        default:
            delete old;
            break;
    }
}

// test WCValskipList<int>
void test_val_skip_list() {
    WCValskipList<int> skiplist( WCSKIPLIST_PROB_QUARTER
                                , WCDEFAULT_SKIPLIST_MAX_PTRS
                                , &val_skip_list_alloc
                                , &val_skip_list_dealloc );

    for( int i = 0; i < NumTestElems; i++ ) {
        skiplist.insert( test_elems[ i ] );
    }

    WCValskipListIter<int> iter( skiplist );
    while( ++iter ) {
        cout << iter.current() << " ";
    }
    cout << "\n\n";
    skiplist.clear();
}
```





---

# ***10 Hash Containers***

This chapter describes hash containers.

**Declared:** `wc-hash.h`

The `WCPtrHashDict<Key, Value>` class is a templated class used to store objects in a dictionary. Dictionaries store values with an associated key, which may be of any type. One example of a dictionary used in everyday life is the phone book. The phone numbers are the data values, and the customer name is the key. An example of a specialized dictionary is a vector, where the key value is the integer index.

As an element is looked up or inserted into the dictionary, the associated key is hashed. Hashing converts the key into a numeric index value which is used to locate the value. The storage area referenced by the hash value is usually called a bucket. If more than one key results in the same hash, the values associated with the keys are placed in a list stored in the bucket. The equality operator of the key's type is used to locate the key-value pairs.

In the description of each member function, the text `Key` is used to indicate the template parameter defining the type of the indices pointed to by the pointers stored in the dictionary. The text `Value` is used to indicate the template parameter defining the type of the data pointed to by the pointers stored in the dictionary.

The constructor for the `WCPtrHashDict<Key, Value>` class requires a hashing function, which given a reference to `Key`, returns an unsigned value. The returned value modulo the number of buckets determines the bucket into which the key-value pair will be located. The return values of the hash function can be spread over the entire range of unsigned numbers. The hash function return value must be the same for values which are equivalent by the equivalence operator for `Key`.

Note that pointers to the key values are stored in the dictionary. Destructors are not called on the keys pointed to. The key values pointed to in the dictionary should not be changed such that the equivalence to the old value is modified.

The `WCEXcept` class is a base class of the `WCPtrHashDict<Key, Value>` class and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCPtrHashDict<Key, Value>` object. No exceptions are enabled unless they are set by the `exceptions` member function.

### Requirements of Key

The `WCPtrHashDict<Key, Value>` class requires `Key` to have:

A well defined equivalence operator with constant parameters  
(`int operator ==( const Key & ) const`).

### Public Member Functions

The following member functions are declared in the public interface:

```
WCPtrHashDict( unsigned (*hash_fn)( const Key & ), unsigned =
WC_DEFAULT_HASH_SIZE );
WCPtrHashDict( unsigned (*hash_fn)( const Key & ), unsigned =
WC_DEFAULT_HASH_SIZE, void * (*user_alloc)( size_t size ),
void (*user_dealloc)( void *old, size_t size ) );
WCPtrHashDict( const WCPtrHashDict & );
virtual ~WCPtrHashDict();
static unsigned bitHash( const void *, size_t );
unsigned buckets() const;
void clear();
void clearAndDestroy();
int contains( const Key * ) const;
unsigned entries() const;
Value * find( const Key * ) const;
Value * findKeyAndValue( const Key *, Key * & ) const;
void forAll( void (*user_fn)( Key *, Value *, void * ) , void
* );
int insert( Key *, Value * );
int isEmpty() const;
Value * remove( const Key * );
void resize( unsigned );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
Value * & operator []( const Key & );
const Value * & operator []( const Key & ) const;
WCPtrHashDict & operator =( const WCPtrHashDict & );
int operator ==( const WCPtrHashDict & ) const;
```

## ***WCPtrHashDict<Key,Value>::WCPtrHashDict()***

---

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashDict( unsigned (*hash_fn)( const Key & ),
unsigned = WC_DEFAULT_HASH_SIZE );
```

**Semantics:** The public `WCPtrHashDict<Key,Value>` constructor creates an `WCPtrHashDict<Key,Value>` object with no entries and with the number of buckets in the second optional parameter, which defaults to the constant `WC_DEFAULT_HASH_SIZE` (currently defined as 101). The number of buckets specified must be greater than zero, and will be forced to at least one. If the hash dictionary object can be created, but an allocation failure occurs when creating the buckets, the table will be created with zero buckets. If the `out_of_memory` exception is enabled, then attempting to insert into a hash table with zero buckets will throw an `out_of_memory` error.

The hash function `hash_fn` is used to determine which bucket each key-value pair will be assigned. If no hash function exists, the static member function `bitHash` is available to help create one.

**Results:** The public `WCPtrHashDict<Key,Value>` constructor creates an initialized `WCPtrHashDict<Key,Value>` object with the specified number of buckets and hash function.

**See Also:** `~WCPtrHashDict`, `bitHash`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashDict( unsigned (*hash_fn)( const Key & ),
unsigned = WC_DEFAULT_HASH_SIZE,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the hash dictionary. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a hash dictionary. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCPtrHashDictItemSize( Key, Value )
```

**Results:** The public `WCPtrHashDict<Key, Value>` constructor creates an initialized `WCPtrHashDict<Key, Value>` object with the specified number of buckets and hash function.

**See Also:** `~WCPtrHashDict`, `bitHash`, `WCExcept::out_of_memory`

## ***WPtrHashDict<Key,Value>::WPtrHashDict()***

---

**Synopsis:**

```
#include <wchash.h>
public:
    WPtrHashDict( const WPtrHashDict & );
```

**Semantics:** The public `WPtrHashDict<Key,Value>` constructor is the copy constructor for the `WPtrHashDict<Key,Value>` class. The new dictionary is created with the same number of buckets, hash function, all values or pointers stored in the dictionary, and the exception trap states. If the hash dictionary object can be created, but an allocation failure occurs when creating the buckets, the table will be created with zero buckets. If there is not enough memory to copy all of the values in the dictionary, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The public `WPtrHashDict<Key,Value>` constructor creates an `WPtrHashDict<Key,Value>` object which is a copy of the passed dictionary.

**See Also:** `~WPtrHashDict`, `operator =`, `WExcept::out_of_memory`

**Synopsis:**

```
#include <wchash.h>
public:
virtual ~WPtrHashDict();
```

**Semantics:** The public `~WPtrHashDict<Key, Value>` destructor is the destructor for the `WPtrHashDict<Key, Value>` class. If the number of dictionary elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the dictionary elements are cleared using the `clear` member function. The objects which the dictionary elements point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the public `~WPtrHashDict<Key, Value>` destructor is inserted implicitly by the compiler at the point where the `WPtrHashDict<Key, Value>` object goes out of scope.

**Results:** The public `~WPtrHashDict<Key, Value>` destructor destroys an `WPtrHashDict<Key, Value>` object.

**See Also:** `clear`, `clearAndDestroy`, `WExcept::not_empty`

## ***WCPtrHashDict<Key,Value>::bitHash()***

---

**Synopsis:**

```
#include <wchash.h>
public:
static unsigned bitHash( void *, size_t );
```

**Semantics:** The `bitHash` public member function can be used to implement a hashing function for any type. A hashing value is generated from the value stored for the number of specified bytes pointed to by the first parameter.

**Results:** The `bitHash` public member function returns an unsigned value which can be used as the basis of a user defined hash function.

**See Also:** `WCPtrHashDict`



**Synopsis:**

```
#include <wchash.h>
public:
    unsigned buckets const;
```

**Semantics:** The `buckets` public member function is used to find the number of buckets contained in the `WCPtrHashDict<Key,Value>` object.

**Results:** The `buckets` public member function returns the number of buckets in the dictionary.

**See Also:** `resize`

## ***WCPtrHashDict<Key,Value>::clear()***

---

**Synopsis:**

```
#include <wchash.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the dictionary so that it has no entries. The number of buckets remain unaffected. Objects pointed to by the dictionary elements are not deleted. The dictionary object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the dictionary to have no elements.

**See Also:** `~WCPtrHashDict`, `clearAndDestroy`, `operator =`

**Synopsis:**

```
#include <wchash.h>
public:
void clearAndDestroy();
```

**Semantics:** The `clearAndDestroy` public member function is used to clear the dictionary and delete the objects pointed to by the dictionary elements. The dictionary object is not destroyed and re-created by this function, so the dictionary object destructor is not invoked.

**Results:** The `clearAndDestroy` public member function clears the dictionary by deleting the objects pointed to by the dictionary elements.

**See Also:** `clear`

## ***WCPtrHashDict<Key,Value>::contains()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int contains( const Key * ) const;
```

**Semantics:** The `contains` public member function returns non-zero if an element with the specified key is stored in the dictionary, or zero if there is no equivalent element. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The `contains` public member function returns a non-zero value if the `Key` is found in the dictionary.

**See Also:** `find`, `findKeyAndValue`

**Synopsis:**

```
#include <wchash.h>
public:
    unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to return the current number of elements stored in the dictionary.

**Results:** The `entries` public member function returns the number of elements in the dictionary.

**See Also:** `buckets`, `isEmpty`

## *WCPtrHashDict<Key,Value>::find()*

---

**Synopsis:**

```
#include <wchash.h>
public:
Value * find( const Key * ) const;
```

**Semantics:** The `find` public member function is used to find an element with an equivalent key in the dictionary. If an equivalent element is found, a pointer to the element `Value` is returned. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element equivalent to the passed key is located in the dictionary.

**See Also:** `findKeyAndValue`

**Synopsis:**

```
#include <wchash.h>
public:
Value * findKeyAndValue( const Key *,
Key &, Value & ) const;
```

**Semantics:** The `findKeyAndValue` public member function is used to find an element in the dictionary with an key equivalent to the first parameter. If an equivalent element is found, a pointer to the element `Value` is returned. The reference to a `Key` passed as the second parameter is assigned the found element's key. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element equivalent to the passed key is located in the dictionary.

**See Also:** `findKeyAndValue`

## ***WCPtrHashDict<Key,Value>::forall()***

---

**Synopsis:**

```
#include <wchash.h>
public:
void forall(
void (*user_fn)( Key *, Value *, void * ),
void * );
```

**Semantics:** The `forall` public member function causes the user supplied function to be invoked for every key-value pair in the dictionary. The user function has the prototype

```
void user_func( Key * key, Value * value, void * data );
```

As the elements are visited, the user function is invoked with the `Key` and `Value` components of the element passed as the first two parameters. The second parameter of the `forall` function is passed as the third parameter to the user function. This value can be used to pass any appropriate data from the main code to the user function.

**Results:** The elements in the dictionary are all visited, with the user function being invoked for each one.

**See Also:** `find`, `findKeyAndValue`



**Synopsis:**

```
#include <wchash.h>
public:
int insert( Key *, Value * );
```

**Semantics:** The `insert` public member function inserts a key and value into the dictionary, using the hash function on the key to determine to which bucket it should be stored. If allocation of the node to store the key-value pair fails, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

At some point, the number of buckets initially selected may be too small for the number of elements inserted. The resize of the dictionary can be controlled by the insertion mechanism by using `WCPtrHashDict` as a base class, and providing an `insert` member function to do a resize when appropriate. This insert could then call `WCPtrHashDict::insert` to insert the element. Note that copy constructors and assignment operators are not inherited in your class, but you can provide the following inline definitions (assuming that the class inherited from `WCPtrHashDict` is named `MyHashDict`):

```
inline MyHashDict( const MyHashDict &orig ) : WCPtrHashDict( orig ) {};
inline MyHashDict &operator=( const MyHashDict &orig ) {
    return( WCPtrHashDict::operator=( orig ) );
}
```

**Results:** The `insert` public member function inserts a key and value into the dictionary. If the insert is successful, a non-zero will returned. A zero will be returned if the insert fails.

**See Also:** `operator =`, `WCEexcept::out_of_memory`

## ***WCPtrHashDict<Key,Value>::isEmpty()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if the dictionary is empty.

**Results:** The `isEmpty` public member function returns zero if it contains at least one entry, non-zero if the dictionary is empty.

**See Also:** `buckets`, `entries`

**Synopsis:**

```
#include <wchash.h>
public:
Value * & operator [] ( const Key & );
```

**Semantics:** `operator []` is the dictionary index operator. A reference to the object stored in the dictionary with the given `Key` is returned. If no equivalent element is found, then a new key-value pair is created with the specified `Key` value, and initialized with the default constructor. The returned reference can then be assigned to, so that insertions can be made with the operator. If an allocation error occurs while inserting a new key-value pair, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, then a reference to address zero will be returned. This will result in a run-time error on systems which trap address zero references.

**Results:** The `operator []` public member function returns a reference to the element at the given key value. If the key does not exist, a reference to a created element is returned. The result of the operator may be assigned to.

**See Also:** `WCEexcept::out_of_memory`

## ***WCPtrHashDict<Key,Value>::operator []()***

---

**Synopsis:**

```
#include <wchash.h>
public:
Value * const & operator []( const Key * ) const;
```

**Semantics:** `operator []` is the dictionary index operator. A constant reference to the object stored in the dictionary with the given `Key` is returned. If no equivalent element is found, then the `index_range` exception is thrown if it is enabled. If the exception is not enabled, then a reference to address zero will be returned. This will result in a run-time error on systems which trap address zero references.

**Results:** The `operator []` public member function returns a constant reference to the element at the given key value. The result of the operator may not be assigned to.

**See Also:** `WCEXcept::index_range`

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashDict & operator =( const WCPtrHashDict & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCPtrHashDict<Key, Value>` class. The left hand side dictionary is first cleared using the `clear` member function, and then the right hand side dictionary is copied. The hash function, exception trap states, and all of the dictionary elements are copied. If an allocation failure occurs when creating the buckets, the table will be created with zero buckets, and the `out_of_memory` exception is thrown if it is enabled. If there is not enough memory to copy all of the values or pointers in the dictionary, then only some will be copied, and the `out_of_memory` exception is thrown if it is enabled. The number of entries will correctly reflect the number copied.

**Results:** The `operator =` public member function assigns the left hand side dictionary to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

## ***WCPtrHashDict<Key,Value>::operator ==( )***

---

**Synopsis:** `#include <wchash.h>`  
`public:`  
`int operator ==( const WCPtrHashDict & ) const;`

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCPtrHashDict<Key,Value>` class. Two dictionary objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side dictionary are the same object. A FALSE (zero) value is returned otherwise.

**Synopsis:**

```
#include <wchash.h>
public:
Value * remove( const Key * );
```

**Semantics:** The `remove` public member function is used to remove the specified element from the dictionary. If an equivalent element is found, the pointer value is returned. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element is removed from the dictionary if it found.

## ***WCPtrHashDict<Key,Value>::resize()***

---

**Synopsis:**

```
#include <wchash.h>
public:
void resize( unsigned );
```

**Semantics:** The `resize` public member function is used to change the number of buckets contained in the dictionary. If the new number is larger than the previous dictionary size, then the hash function will be used on all of the stored elements to determine which bucket they should be stored into. Entries are not destroyed or created in the process of being moved. If there is not enough memory to resize the dictionary, the `out_of_memory` exception is thrown if it is enabled, and the dictionary will contain the number of buckets it contained before the resize. If the new number is zero, then the `zero_buckets` exception is thrown if it is enabled, and no resize will be performed. The dictionary is guaranteed to contain the same number of entries after the resize.

**Results:** The dictionary is resized to the new number of buckets.

**See Also:** `WCEexcept::out_of_memory`, `WCEexcept::zero_buckets`



**Declared:** `wc-hash.h`

`WCPtrHashTable<Type>` and `WCPtrHashSet<Type>` classes are templated classes used to store objects in a hash. A hash saves objects in such a way as to make it efficient to locate and retrieve an element. As an element is looked up or inserted into the hash, the value of the element is hashed. Hashing results in a numeric index which is used to locate the value. The storage area referenced by the hash value is usually called a bucket. If more than one element results in the same hash, the value associated with the hash is placed in a list stored in the bucket. A hash table allows more than one copy of an element that is equivalent, while the hash set allows only one copy. The equality operator of the element's type is used to locate the value.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type of the data pointed to by the pointers stored in the hash.

The constructor for the `WCPtrHashTable<Type>` and `WCPtrHashSet<Type>` classes requires a hashing function, which given a reference to `Type`, returns an unsigned value. The returned value modulo the number of buckets determines the bucket into which the element will be located. The return values of the hash function can be spread over the entire range of unsigned numbers. The hash function return value must be the same for values which are equivalent by the equivalence operator for `Type`.

Note that pointers to the elements are stored in the hash. Destructors are not called on the elements pointed to. The data values pointed to in the hash should not be changed such that the equivalence to the old value is modified.

The `WCExcept` class is a base class of the `WCPtrHashTable<Type>` and `WCPtrHashSet<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCPtrHashTable<Type>` and `WCPtrHashSet<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Requirements of Type**

The `WCPtrHashTable<Type>` and `WCPtrHashSet<Type>` classes requires `Type` to have:

A well defined equivalence operator with constant parameters  
(`int operator ==(const Type & ) const`).

### **Public Member Functions**

The following member functions are declared in the public interface:

## ***WCPtrHashTable<Type>, WCPtrHashSet<Type>***

---

```
WCPtrHashSet( unsigned (*hash_fn)( const Type & ), unsigned =
WC_DEFAULT_HASH_SIZE );
WCPtrHashSet( unsigned (*hash_fn)( const Type & ), unsigned =
WC_DEFAULT_HASH_SIZE, void * (*user_alloc)( size_t size ),
void (*user_dealloc)( void *old, size_t size ) );
WCPtrHashSet( const WCPtrHashSet & );
virtual ~WCPtrHashSet();
WCPtrHashTable( unsigned (*hash_fn)( const Type & ), unsigned =
WC_DEFAULT_HASH_SIZE );
WCPtrHashTable( unsigned (*hash_fn)( const Type & ), unsigned =
WC_DEFAULT_HASH_SIZE, void * (*user_alloc)( size_t size ),
void (*user_dealloc)( void *old, size_t size ) );
WCPtrHashTable( const WCPtrHashTable & );
virtual ~WCPtrHashTable();
static unsigned bitHash( const void *, size_t );
unsigned buckets() const;
void clear();
void clearAndDestroy();
int contains( const Type * ) const;
unsigned entries() const;
Type * find( const Type * ) const;
void forAll( void (*user_fn)( Type *, void * ) , void * );
int insert( Type * );
int isEmpty() const;
Type * remove( const Type * );
void resize( unsigned );
```

The following public member functions are available for the `WCPtrHashTable` class only:

```
unsigned occurrencesOf( const Type * ) const;
unsigned removeAll( const Type * );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
WCPtrHashSet & operator =( const WCPtrHashSet & );
int operator ==( const WCPtrHashSet & ) const;
WCPtrHashTable & operator =( const WCPtrHashTable & );
int operator ==( const WCPtrHashTable & ) const;
```

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashSet( unsigned (*hash_fn)( const Type & ),
unsigned = WC_DEFAULT_HASH_SIZE );
```

**Semantics:** The `WCPtrHashSet<Type>` constructor creates a `WCPtrHashSet` object with no entries and with the number of buckets in the second optional parameter, which defaults to the constant `WC_DEFAULT_HASH_SIZE` (currently defined as 101). The number of buckets specified must be greater than zero, and will be forced to at least one. If the hash object can be created, but an allocation failure occurs when creating the buckets, the table will be created with zero buckets. If the `out_of_memory` exception is enabled, then attempting to insert into a hash table with zero buckets will throw an `out_of_memory` error.

The hash function `hash_fn` is used to determine which bucket each value will be assigned to. If no hash function exists, the static member function `bitHash` is available to help create one.

**Results:** The `WCPtrHashSet<Type>` constructor creates an initialized `WCPtrHashSet` object with the specified number of buckets and hash function.

**See Also:** `~WCPtrHashSet`, `bitHash`, `WCEXCEPT::out_of_memory`

## ***WCPtrHashSet<Type>::WCPtrHashSet()***

---

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashSet( unsigned (*hash_fn)( const Type & ),
unsigned = WC_DEFAULT_HASH_SIZE,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the hash. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a hash. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCPtrHashSetItemSize( Type )
```

**Results:** The `WCPtrHashSet<Type>` constructor creates an initialized `WCPtrHashSet` object with the specified number of buckets and hash function.

**See Also:** `~WCPtrHashSet`, `bitHash`, `WCEexcept::out_of_memory`

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashSet( const WCPtrHashSet & );
```

**Semantics:** The `WCPtrHashSet<Type>` is the copy constructor for the `WCPtrHashSet` class. The new hash is created with the same number of buckets, hash function, all values or pointers stored in the hash, and the exception trap states. If the hash object can be created, but an allocation failure occurs when creating the buckets, the hash will be created with zero buckets. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The `WCPtrHashSet<Type>` constructor creates a `WCPtrHashSet` object which is a copy of the passed hash.

**See Also:** `~WCPtrHashSet`, `operator =`, `WExcept::out_of_memory`

## ***WCPtrHashSet<Type>::~~WCPtrHashSet()***

---

**Synopsis:**

```
#include <wchash.h>
public:
virtual ~WCPtrHashSet();
```

**Semantics:** The `WCPtrHashSet<Type>` destructor is the destructor for the `WCPtrHashSet` class. If the number of elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the hash elements are cleared using the `clear` member function. The objects which the hash elements point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the `WCPtrHashSet<Type>` destructor is inserted implicitly by the compiler at the point where the `WCPtrHashSet` object goes out of scope.

**Results:** The call to the `WCPtrHashSet<Type>` destructor destroys a `WCPtrHashSet` object.

**See Also:** `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashTable( unsigned (*hash_fn)( const Type & ),
unsigned = WC_DEFAULT_HASH_SIZE );
```

**Semantics:** The `WCPtrHashTable<Type>` constructor creates a `WCPtrHashTable` object with no entries and with the number of buckets in the second optional parameter, which defaults to the constant `WC_DEFAULT_HASH_SIZE` (currently defined as 101). The number of buckets specified must be greater than zero, and will be forced to at least one. If the hash object can be created, but an allocation failure occurs when creating the buckets, the table will be created with zero buckets. If the `out_of_memory` exception is enabled, then attempting to insert into a hash table with zero buckets will throw an `out_of_memory` error.

The hash function `hash_fn` is used to determine which bucket each value will be assigned to. If no hash function exists, the static member function `bitHash` is available to help create one.

**Results:** The `WCPtrHashTable<Type>` constructor creates an initialized `WCPtrHashTable` object with the specified number of buckets and hash function.

**See Also:** `~WCPtrHashTable`, `bitHash`, `WCEXCEPT::out_of_memory`

## ***WCPtrHashTable<Type>::WCPtrHashTable()***

---

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashTable( unsigned (*hash_fn)( const Type & ),
unsigned = WC_DEFAULT_HASH_SIZE,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the hash. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a hash. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCPtrHashTableItemSize( Type )
```

**Results:** The `WCPtrHashTable<Type>` constructor creates an initialized `WCPtrHashTable` object with the specified number of buckets and hash function.

**See Also:** `~WCPtrHashTable`, `bitHash`, `WCEXCEPT::out_of_memory`



**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashTable( const WCPtrHashTable & );
```

**Semantics:** The `WCPtrHashTable<Type>` is the copy constructor for the `WCPtrHashTable` class. The new hash is created with the same number of buckets, hash function, all values or pointers stored in the hash, and the exception trap states. If the hash object can be created, but an allocation failure occurs when creating the buckets, the hash will be created with zero buckets. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The `WCPtrHashTable<Type>` constructor creates a `WCPtrHashTable` object which is a copy of the passed hash.

**See Also:** `~WCPtrHashTable`, `operator =`, `WCEXCEPT::out_of_memory`

## ***WCPtrHashTable<Type>::~~WCPtrHashTable()***

---

**Synopsis:**

```
#include <wchash.h>
public:
virtual ~WCPtrHashTable();
```

**Semantics:** The `WCPtrHashTable<Type>` destructor is the destructor for the `WCPtrHashTable` class. If the number of elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the hash elements are cleared using the `clear` member function. The objects which the hash elements point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the `WCPtrHashTable<Type>` destructor is inserted implicitly by the compiler at the point where the `WCPtrHashTable` object goes out of scope.

**Results:** The call to the `WCPtrHashTable<Type>` destructor destroys a `WCPtrHashTable` object.

**See Also:** `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

## *WCPtrHashTable<Type>::bitHash(), WCPtrHashSet<Type>::bitHash()*

---

**Synopsis:**

```
#include <wchash.h>
public:
static unsigned bitHash( void *, size_t );
```

**Semantics:** The `bitHash` public member function can be used to implement a hashing function for any type. A hashing value is generated from the value stored for the number of specified bytes pointed to by the first parameter.

**Results:** The `bitHash` public member function returns an unsigned value which can be used as the basis of a user defined hash function.

**See Also:** `WCPtrHashSet`, `WCPtrHashTable`

## ***WPtrHashTable<Type>::buckets(), WPtrHashSet<Type>::buckets()***

---

**Synopsis:**

```
#include <wchash.h>
public:
    unsigned buckets() const;
```

**Semantics:** The `buckets` public member function is used to find the number of buckets contained in the hash object.

**Results:** The `buckets` public member function returns the number of buckets in the hash.

**See Also:** `resize`

**Synopsis:**

```
#include <wchash.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the hash so that it has no entries. The number of buckets remain unaffected. Objects pointed to by the hash elements are not deleted. The hash object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the hash to have no elements.

**See Also:** `~WPtrHashSet`, `~WPtrHashTable`, `clearAndDestroy`, `operator =`

## ***WCPtrHashTable<Type>,WCPtrHashSet<Type>::clearAndDestroy()***

---

**Synopsis:** `#include <wchash.h>`  
`public:`  
`void clearAndDestroy();`

**Semantics:** The `clearAndDestroy` public member function is used to clear the hash and delete the objects pointed to by the hash elements. The hash object is not destroyed and re-created by this function, so the hash object destructor is not invoked.

**Results:** The `clearAndDestroy` public member function clears the hash by deleting the objects pointed to by the hash elements.

**See Also:** `clear`

## *WPtrHashTable<Type>::contains(), WPtrHashSet<Type>::contains()*

---

**Synopsis:**

```
#include <wchash.h>
public:
int contains( const Type * ) const;
```

**Semantics:** The `contains` public member function returns non-zero if the element is stored in the hash, or zero if there is no equivalent element. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The `contains` public member function returns a non-zero value if the element is found in the hash.

**See Also:** `find`

## ***WPtrHashTable<Type>::entries(), WPtrHashSet<Type>::entries()***

---

**Synopsis:**

```
#include <wchash.h>
public:
    unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to return the current number of elements stored in the hash.

**Results:** The `entries` public member function returns the number of elements in the hash.

**See Also:** `buckets`, `isEmpty`



**Synopsis:**

```
#include <wchash.h>
public:
Type * find( const Type * ) const;
```

**Semantics:** The `find` public member function is used to find an element with an equivalent key in the hash. If an equivalent element is found, a pointer to the element is returned. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The element equivalent to the passed key is located in the hash.

## ***WCPtrHashTable<Type>::forAll(), WCPtrHashSet<Type>::forAll()***

---

**Synopsis:**

```
#include <wchash.h>
public:
void forAll(
void (*user_fn)( Type *, void * ),
void * );
```

**Semantics:** The `forAll` public member function causes the user supplied function to be invoked for every value in the hash. The user function has the prototype

```
void user_func( Type * value, void * data );
```

As the elements are visited, the user function is invoked with the element passed as the first. The second parameter of the `forAll` function is passed as the second parameter to the user function. This value can be used to pass any appropriate data from the main code to the user function.

**Results:** The elements in the hash are all visited, with the user function being invoked for each one.

**See Also:** `find`

**Synopsis:**

```
#include <whash.h>
public:
int insert( Type * );
```

**Semantics:** The `insert` public member function inserts a value into the hash, using the hash function to determine to which bucket it should be stored. If allocation of the node to store the value fails, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

With a `WCPtrHashSet`, there must be only one equivalent element in the set. If an element equivalent to the inserted element is already in the hash set, the hash set will remain unchanged, and the `not_unique` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

At some point, the number of buckets initially selected may be too small for the number of elements inserted. The `resize` of the hash can be controlled by the insertion mechanism by using `WCPtrHashSet` (or `WCPtrHashTable`) as a base class, and providing an `insert` member function to do a `resize` when appropriate. This `insert` could then call `WCPtrHashSet::insert` (or `WCPtrHashTable::insert`) to insert the element. Note that copy constructors and assignment operators are not inherited in your class, but you can provide the following inline definitions (assuming that the class inherited from `WCPtrHashTable` is named `MyHashTable`):

```
inline MyHashTable( const MyHashTable &orig )
    : WCPtrHashTable( orig ) {};
inline MyHashTable &operator=( const MyHashTable &orig ) {
    return( WCPtrHashTable::operator=( orig ) );
}
```

**Results:** The `insert` public member function inserts a value into the hash. If the insert is successful, a non-zero will returned. A zero will be returned if the insert fails.

**See Also:** `operator =`, `WCExcept::out_of_memory`

## ***WCPtrHashTable<Type>::isEmpty(), WCPtrHashSet<Type>::isEmpty()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if the hash is empty.

**Results:** The `isEmpty` public member function returns zero if it contains at least one entry, non-zero if the hash is empty.

**See Also:** `buckets`, `entries`

**Synopsis:**

```
#include <wchash.h>
public:
    unsigned occurrencesOf( const Type * ) const;
```

**Semantics:** The `occurrencesOf` public member function is used to return the current number of elements stored in the hash which are equivalent to the passed value. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The `occurrencesOf` public member function returns the number of elements in the hash.

**See Also:** `buckets`, `entries`, `find`, `isEmpty`

## ***WCPtrHashTable<Type>::operator =(), WCPtrHashSet<Type>::operator =()***

---

**Synopsis:**

```
#include <wchash.h>
public:
WCPtrHashSet & operator =( const WCPtrHashSet & );
WCPtrHashTable & operator =( const WCPtrHashTable & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCPtrHashTable<Type>` and `WCPtrHashSet<Type>` classes. The left hand side hash is first cleared using the `clear` member function, and then the right hand side hash is copied. The hash function, exception trap states, and all of the hash elements are copied. If an allocation failure occurs when creating the buckets, the table will be created with zero buckets, and the `out_of_memory` exception is thrown if it is enabled. If there is not enough memory to copy all of the values or pointers in the hash, then only some will be copied, and the `out_of_memory` exception is thrown if it is enabled. The number of entries will correctly reflect the number copied.

**Results:** The `operator =` public member function assigns the left hand side hash to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

## ***WCPtrHashTable<Type>::operator ==( ), WCPtrHashSet<Type>::operator ==( )***

---

**Synopsis:**

```
#include <wchash.h>
public:
int operator ==( const WCPtrHashSet & ) const;
int operator ==( const WCPtrHashTable & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCPtrHashTable<Type>` and `WCPtrHashSet<Type>` classes. Two hash objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side hash are the same object. A FALSE (zero) value is returned otherwise.

## ***WCPtrHashTable<Type>::remove(), WCPtrHashSet<Type>::remove()***

---

**Synopsis:**

```
#include <wchash.h>
public:
Type * remove( const Type * );
```

**Semantics:** The `remove` public member function is used to remove the specified element from the hash. If an equivalent element is found, the pointer value is returned. Zero is returned if the element is not found. If the hash is a table and there is more than one element equivalent to the specified element, then the first equivalent element added to the table is removed. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The element is removed from the hash if it found.



**Synopsis:**

```
#include <wchash.h>
public:
unsigned removeAll( const Type * );
```

**Semantics:** The `removeAll` public member function is used to remove all elements equivalent to the specified element from the hash. Zero is returned if no equivalent elements are found. Note that equivalence is based on the equivalence operator of the element type.

**Results:** All equivalent elements are removed from the hash.

## ***WCPtrHashTable<Type>::resize(), WCPtrHashSet<Type>::resize()***

---

**Synopsis:**

```
#include <wchash.h>
public:
void resize( unsigned );
```

**Semantics:** The `resize` public member function is used to change the number of buckets contained in the hash. If the new number is larger than the previous hash size, then the hash function will be used on all of the stored elements to determine which bucket they should be stored into. Entries are not destroyed or created in the process of being moved. If there is not enough memory to resize the hash, the `out_of_memory` exception is thrown if it is enabled, and the hash will contain the number of buckets it contained before the resize. If the new number is zero, then the `zero_buckets` exception is thrown if it is enabled, and no resize will be performed. The hash is guaranteed to contain the same number of entries after the resize.

**Results:** The hash is resized to the new number of buckets.

**See Also:** `WCEexcept::out_of_memory`, `WCEexcept::zero_buckets`

**Declared:** wchash.h

The `WCValHashDict<Key, Value>` class is a templated class used to store objects in a dictionary. Dictionaries store values with an associated key, which may be of any type. One example of a dictionary used in everyday life is the phone book. The phone numbers are the data values, and the customer name is the key. An example of a specialized dictionary is a vector, where the key value is the integer index.

As an element is looked up or inserted into the dictionary, the associated key is hashed. Hashing converts the key into a numeric index value which is used to locate the value. The storage area referenced by the hash value is usually called a bucket. If more than one key results in the same hash, the values associated with the keys are placed in a list stored in the bucket. The equality operator of the key's type is used to locate the key-value pairs.

In the description of each member function, the text `Key` is used to indicate the template parameter defining the type of the indices used to store data in the dictionary. The text `Value` is used to indicate the template parameter defining the type of the data stored in the dictionary.

The constructor for the `WCValHashDict<Key, Value>` class requires a hashing function, which given a reference to `Key`, returns an unsigned value. The returned value modulo the number of buckets determines the bucket into which the key-value pair will be located. The return values of the hash function can be spread over the entire range of unsigned numbers. The hash function return value must be the same for values which are equivalent by the equivalence operator for `Key`.

Values are copied into the dictionary, which could be undesirable if the stored objects are complicated and copying is expensive. Value dictionaries should not be used to store objects of a base class if any derived types of different sizes would be stored in the dictionary, or if the destructor for a derived class must be called.

The `WCExcept` class is a base class of the `WCValHashDict<Key, Value>` class and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCValHashDict<Key, Value>` object. No exceptions are enabled unless they are set by the `exceptions` member function.

### Requirements of Key and Value

The `WCValHashDict<Key, Value>` class requires `Key` to have:

A default constructor (`Key::Key()`).

A well defined copy constructor (`Key::Key(const Key &)`).

A well defined assignment operator ( Key & operator =( const Key & ) ).

A well defined equivalence operator with constant parameters  
( int operator ==( const Key & ) const ).

The WCValHashDict<Key, Value> class requires Value to have:

A default constructor ( Value::Value() ).

A well defined copy constructor ( Value::Value( const Value & ) ).

A well defined assignment operator ( Value & operator =( const Value & ) ).

### Public Member Functions

The following member functions are declared in the public interface:

```
WCValHashDict( unsigned (*hash_fn)( const Key & ), unsigned =
WC_DEFAULT_HASH_SIZE );
WCValHashDict( unsigned (*hash_fn)( const Key & ), unsigned =
WC_DEFAULT_HASH_SIZE, void * (*user_alloc)( size_t size ),
void (*user_dealloc)( void *old, size_t size ) );
WCValHashDict( const WCValHashDict & );
virtual ~WCValHashDict();
static unsigned bitHash( const void *, size_t );
unsigned buckets() const;
void clear();
int contains( const Key & ) const;
unsigned entries() const;
int find( const Key &, Value & ) const;
int findKeyAndValue( const Key &, Key &, Value & ) const;
void forAll( void (*user_fn)( Key, Value, void * ), void * );
int insert( const Key &, const Value & );
int isEmpty() const;
int remove( const Key & );
void resize( unsigned );
```

### Public Member Operators

The following member operators are declared in the public interface:

```
Value & operator [] ( const Key & );
const Value & operator [] ( const Key & ) const;
WCValHashDict & operator =( const WCValHashDict & );
int operator ==( const WCValHashDict & ) const;
```

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashDict( unsigned (*hash_fn)( const Key & ),
unsigned = WC_DEFAULT_HASH_SIZE );
```

**Semantics:** The public `WCValHashDict<Key, Value>` constructor creates an `WCValHashDict<Key, Value>` object with no entries and with the number of buckets in the second optional parameter, which defaults to the constant `WC_DEFAULT_HASH_SIZE` (currently defined as 101). The number of buckets specified must be greater than zero, and will be forced to at least one. If the hash dictionary object can be created, but an allocation failure occurs when creating the buckets, the table will be created with zero buckets. If the `out_of_memory` exception is enabled, then attempting to insert into a hash table with zero buckets will throw an `out_of_memory` error.

The hash function `hash_fn` is used to determine which bucket each key-value pair will be assigned. If no hash function exists, the static member function `bitHash` is available to help create one.

**Results:** The public `WCValHashDict<Key, Value>` constructor creates an initialized `WCValHashDict<Key, Value>` object with the specified number of buckets and hash function.

**See Also:** `~WCValHashDict`, `bitHash`, `WCEXCEPT::out_of_memory`

## ***WCValHashDict<Key, Value>::WCValHashDict()***

---

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashDict( unsigned (*hash_fn)( const Key & ),
unsigned = WC_DEFAULT_HASH_SIZE,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the hash dictionary. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a hash dictionary. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCValHashDictItemSize( Key, Value )
```

**Results:** The public `WCValHashDict<Key, Value>` constructor creates an initialized `WCValHashDict<Key, Value>` object with the specified number of buckets and hash function.

**See Also:** `~WCValHashDict`, `bitHash`, `WCExcept::out_of_memory`

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashDict( const WCValHashDict & );
```

**Semantics:** The public `WCValHashDict<Key, Value>` constructor is the copy constructor for the `WCValHashDict<Key, Value>` class. The new dictionary is created with the same number of buckets, hash function, all values or pointers stored in the dictionary, and the exception trap states. If the hash dictionary object can be created, but an allocation failure occurs when creating the buckets, the table will be created with zero buckets. If there is not enough memory to copy all of the values in the dictionary, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The public `WCValHashDict<Key, Value>` constructor creates an `WCValHashDict<Key, Value>` object which is a copy of the passed dictionary.

**See Also:** `~WCValHashDict`, `operator =`, `WCEXCEPT::out_of_memory`

## ***WCValHashDict<Key, Value>::~~WCValHashDict()***

---

**Synopsis:**

```
#include <wchash.h>
public:
virtual ~WCValHashDict();
```

**Semantics:** The public `~WCValHashDict<Key, Value>` destructor is the destructor for the `WCValHashDict<Key, Value>` class. If the number of dictionary elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the dictionary elements are cleared using the `clear` member function. The call to the public `~WCValHashDict<Key, Value>` destructor is inserted implicitly by the compiler at the point where the `WCValHashDict<Key, Value>` object goes out of scope.

**Results:** The public `~WCValHashDict<Key, Value>` destructor destroys an `WCValHashDict<Key, Value>` object.

**See Also:** `clear`, `WCEXCEPT::not_empty`



**Synopsis:**

```
#include <wchash.h>
public:
static unsigned bitHash( void *, size_t );
```

**Semantics:** The `bitHash` public member function can be used to implement a hashing function for any type. A hashing value is generated from the value stored for the number of specified bytes pointed to by the first parameter. For example:

```
unsigned my_hash_fn( const int & key ) {
    return( WCValHashDict<int,String>::bitHash( &key, sizeof( int ) );
}
WCValHashDict<int,String> data_object( &my_hash_fn );
```

**Results:** The `bitHash` public member function returns an unsigned value which can be used as the basis of a user defined hash function.

**See Also:** `WCValHashDict`

## *WCValHashDict<Key, Value>::buckets()*

---

**Synopsis:**

```
#include <wchash.h>
public:
    unsigned buckets const;
```

**Semantics:** The `buckets` public member function is used to find the number of buckets contained in the `WCValHashDict<Key, Value>` object.

**Results:** The `buckets` public member function returns the number of buckets in the dictionary.

**See Also:** `resize`

**Synopsis:**

```
#include <wchash.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the dictionary so that it has no entries. The number of buckets remain unaffected. Elements stored in the dictionary are destroyed using the destructors of `Key` and of `Value`. The dictionary object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the dictionary to have no elements.

**See Also:** `~WCValHashDict`, `operator =`

## ***WCValHashDict<Key, Value>::contains()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int contains( const Key & ) const;
```

**Semantics:** The `contains` public member function returns non-zero if an element with the specified key is stored in the dictionary, or zero if there is no equivalent element. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The `contains` public member function returns a non-zero value if the `Key` is found in the dictionary.

**See Also:** `find`, `findKeyAndValue`

**Synopsis:**

```
#include <wchash.h>
public:
    unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to return the current number of elements stored in the dictionary.

**Results:** The `entries` public member function returns the number of elements in the dictionary.

**See Also:** `buckets`, `isEmpty`

## *WCValHashDict<Key, Value>::find()*

---

**Synopsis:**

```
#include <wchash.h>
public:
int find( const Key &, Value & ) const;
```

**Semantics:** The `find` public member function is used to find an element with an equivalent key in the dictionary. If an equivalent element is found, a non-zero value is returned. The reference to a `Value` passed as the second argument is assigned the found element's `Value`. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element equivalent to the passed key is located in the dictionary.

**See Also:** `findKeyAndValue`

**Synopsis:**

```
#include <wchash.h>
public:
int findKeyAndValue( const Key &, Key &, Value & ) const;
```

**Semantics:** The `findKeyAndValue` public member function is used to find an element in the dictionary with an key equivalent to the first parameter. If an equivalent element is found, a non-zero value is returned. The reference to a `Key` passed as the second parameter is assigned the found element's key. The reference to a `Value` passed as the third argument is assigned the found element's `Value`. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element equivalent to the passed key is located in the dictionary.

**See Also:** `findKeyAndValue`

## ***WCValHashDict<Key, Value>::forall()***

---

**Synopsis:**

```
#include <wchash.h>
public:
void forall(
void (*user_fn)( Key, Value, void * ),
void * );
```

**Semantics:** The `forall` public member function causes the user supplied function to be invoked for every key-value pair in the dictionary. The user function has the prototype

```
void user_func( Key key, Value value, void * data );
```

As the elements are visited, the user function is invoked with the `Key` and `Value` components of the element passed as the first two parameters. The second parameter of the `forall` function is passed as the third parameter to the user function. This value can be used to pass any appropriate data from the main code to the user function.

**Results:** The elements in the dictionary are all visited, with the user function being invoked for each one.

**See Also:** `find`, `findKeyAndValue`



**Synopsis:**

```
#include <wchash.h>
public:
int insert( const Key &, const Value & );
```

**Semantics:** The `insert` public member function inserts a key and value into the dictionary, using the hash function on the key to determine to which bucket it should be stored. If allocation of the node to store the key-value pair fails, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

At some point, the number of buckets initially selected may be too small for the number of elements inserted. The resize of the dictionary can be controlled by the insertion mechanism by using `WCValHashDict` as a base class, and providing an `insert` member function to do a resize when appropriate. This insert could then call `WCValHashDict::insert` to insert the element. Note that copy constructors and assignment operators are not inherited in your class, but you can provide the following inline definitions (assuming that the class inherited from `WCValHashDict` is named `MyHashDict`):

```
inline MyHashDict( const MyHashDict &orig ) : WCValHashDict( orig ) {};
inline MyHashDict &operator=( const MyHashDict &orig ) {
    return( WCValHashDict::operator=( orig ) );
}
```

**Results:** The `insert` public member function inserts a key and value into the dictionary. If the insert is successful, a non-zero will returned. A zero will be returned if the insert fails.

**See Also:** `operator =`, `WCExcept::out_of_memory`

## ***WCValHashDict<Key, Value>::isEmpty()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if the dictionary is empty.

**Results:** The `isEmpty` public member function returns zero if it contains at least one entry, non-zero if the dictionary is empty.

**See Also:** `buckets`, `entries`

**Synopsis:**

```
#include <wchash.h>
public:
Value & operator [] ( const Key & );
```

**Semantics:** `operator []` is the dictionary index operator. A reference to the object stored in the dictionary with the given `Key` is returned. If no equivalent element is found, then a new key-value pair is created with the specified `Key` value, and initialized with the default constructor. The returned reference can then be assigned to, so that insertions can be made with the operator.

```
WCVaHashDict<int,String> data_object( &my_hash_fn );
data_object[ 5 ] = "Hello";
```

If an allocation error occurs while inserting a new key-value pair, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, then a reference to address zero will be returned. This will result in a run-time error on systems which trap address zero references.

**Results:** The `operator []` public member function returns a reference to the element at the given key value. If the key does not exist, a reference to a created element is returned. The result of the operator may be assigned to.

**See Also:** `WCExcept::out_of_memory`

## ***WCValHashDict<Key, Value>::operator []()***

---

**Synopsis:**

```
#include <wchash.h>
public:
    const Value & operator [] ( const Key & ) const;
```

**Semantics:** `operator []` is the dictionary index operator. A constant reference to the object stored in the dictionary with the given `Key` is returned. If no equivalent element is found, then the `index_range` exception is thrown if it is enabled. If the exception is not enabled, then a reference to address zero will be returned. This will result in a run-time error on systems which trap address zero references.

**Results:** The `operator []` public member function returns a constant reference to the element at the given key value. The result of the operator may not be assigned to.

**See Also:** `WCExcept::index_range`

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashDict & operator =( const WCValHashDict & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCValHashDict<Key, Value>` class. The left hand side dictionary is first cleared using the `clear` member function, and then the right hand side dictionary is copied. The hash function, exception trap states, and all of the dictionary elements are copied. If an allocation failure occurs when creating the buckets, the table will be created with zero buckets, and the `out_of_memory` exception is thrown if it is enabled. If there is not enough memory to copy all of the values or pointers in the dictionary, then only some will be copied, and the `out_of_memory` exception is thrown if it is enabled. The number of entries will correctly reflect the number copied.

**Results:** The `operator =` public member function assigns the left hand side dictionary to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

## ***WCValHashDict<Key,Value>::operator ==()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int operator ==( const WCValHashDict & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCValHashDict<Key,Value>` class. Two dictionary objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side dictionary are the same object. A FALSE (zero) value is returned otherwise.

**Synopsis:**

```
#include <wchash.h>
public:
int remove( const Key & );
```

**Semantics:** The `remove` public member function is used to remove the specified element from the dictionary. If an equivalent element is found, a non-zero value is returned. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element is removed from the dictionary if it found.

## ***WCValHashDict<Key, Value>::resize()***

---

**Synopsis:**

```
#include <wchash.h>
public:
void resize( unsigned );
```

**Semantics:** The `resize` public member function is used to change the number of buckets contained in the dictionary. If the new number is larger than the previous dictionary size, then the hash function will be used on all of the stored elements to determine which bucket they should be stored into. Entries are not destroyed or created in the process of being moved. If there is not enough memory to resize the dictionary, the `out_of_memory` exception is thrown if it is enabled, and the dictionary will contain the number of buckets it contained before the resize. If the new number is zero, then the `zero_buckets` exception is thrown if it is enabled, and no resize will be performed. The dictionary is guaranteed to contain the same number of entries after the resize.

**Results:** The dictionary is resized to the new number of buckets.

**See Also:** `WCExcept::out_of_memory`, `WCExcept::zero_buckets`



**Declared:** `wc-hash.h`

`WCValHashTable<Type>` and `WCValHashSet<Type>` classes are templated classes used to store objects in a hash. A hash saves objects in such a way as to make it efficient to locate and retrieve an element. As an element is looked up or inserted into the hash, the value of the element is hashed. Hashing results in a numeric index which is used to locate the value. The storage area referenced by the hash value is usually called a bucket. If more than one element results in the same hash, the value associated with the hash is placed in a list stored in the bucket. A hash table allows more than one copy of an element that is equivalent, while the hash set allows only one copy. The equality operator of the element's type is used to locate the value.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type of the data to be stored in the hash.

The constructor for the `WCValHashTable<Type>` and `WCValHashSet<Type>` classes requires a hashing function, which given a reference to `Type`, returns an unsigned value. The returned value modulo the number of buckets determines the bucket into which the element will be located. The return values of the hash function can be spread over the entire range of unsigned numbers. The hash function return value must be the same for values which are equivalent by the equivalence operator for `Type`.

Values are copied into the hash, which could be undesirable if the stored objects are complicated and copying is expensive. Value hashes should not be used to store objects of a base class if any derived types of different sizes would be stored in the hash, or if the destructor for a derived class must be called.

The `WCExcept` class is a base class of the `WCValHashTable<Type>` and `WCValHashSet<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCValHashTable<Type>` and `WCValHashSet<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Requirements of Type**

The `WCValHashTable<Type>` and `WCValHashSet<Type>` classes requires `Type` to have:

A default constructor (`Type::Type()`).

A well defined copy constructor (`Type::Type(const Type &)`).

A well defined assignment operator (`Type & operator =(const Type &)`).

## ***WCValHashTable<Type>, WCValHashSet<Type>***

---

A well defined equivalence operator with constant parameters  
(int operator ==( const Type & ) const).

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCValHashSet( unsigned (*hash_fn)( const Type & ), unsigned =  
WC_DEFAULT_HASH_SIZE );  
WCValHashSet( unsigned (*hash_fn)( const Type & ), unsigned =  
WC_DEFAULT_HASH_SIZE, void * (*user_alloc)( size_t size ),  
void (*user_dealloc)( void *old, size_t size ) );  
WCValHashSet( const WCValHashSet & );  
virtual ~WCValHashSet();  
WCValHashTable( unsigned (*hash_fn)( const Type & ), unsigned  
= WC_DEFAULT_HASH_SIZE );  
WCValHashTable( unsigned (*hash_fn)( const Type & ), unsigned  
= WC_DEFAULT_HASH_SIZE, void * (*user_alloc)( size_t size ),  
void (*user_dealloc)( void *old, size_t size ) );  
WCValHashTable( const WCValHashTable & );  
virtual ~WCValHashTable();  
static unsigned bitHash( const void *, size_t );  
unsigned buckets() const;  
void clear();  
int contains( const Type & ) const;  
unsigned entries() const;  
int find( const Type &, Type & ) const;  
void forAll( void (*user_fn)( Type, void * ), void * );  
int insert( const Type & );  
int isEmpty() const;  
int remove( const Type & );  
void resize( unsigned );
```

The following public member functions are available for the WCValHashTable class only:

```
unsigned occurrencesOf( const Type & ) const;  
unsigned removeAll( const Type & );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
WCValHashSet & operator =( const WCValHashSet & );  
int operator ==( const WCValHashSet & ) const;  
WCValHashTable & operator =( const WCValHashTable & );  
int operator ==( const WCValHashTable & ) const;
```

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashSet( unsigned (*hash_fn)( const Type & ),
unsigned = WC_DEFAULT_HASH_SIZE );
```

**Semantics:** The `WCValHashSet<Type>` constructor creates a `WCValHashSet` object with no entries and with the number of buckets in the second optional parameter, which defaults to the constant `WC_DEFAULT_HASH_SIZE` (currently defined as 101). The number of buckets specified must be greater than zero, and will be forced to at least one. If the hash object can be created, but an allocation failure occurs when creating the buckets, the table will be created with zero buckets. If the `out_of_memory` exception is enabled, then attempting to insert into a hash table with zero buckets will throw an `out_of_memory` error.

The hash function `hash_fn` is used to determine which bucket each value will be assigned to. If no hash function exists, the static member function `bitHash` is available to help create one.

**Results:** The `WCValHashSet<Type>` constructor creates an initialized `WCValHashSet` object with the specified number of buckets and hash function.

**See Also:** `~WCValHashSet`, `bitHash`, `WCEXCEPT::out_of_memory`

## ***WCValHashSet<Type>::WCValHashSet()***

---

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashSet( unsigned (*hash_fn)( const Type & ),
unsigned = WC_DEFAULT_HASH_SIZE,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the hash. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a hash. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCValHashSetItemSize( Type )
```

**Results:** The `WCValHashSet<Type>` constructor creates an initialized `WCValHashSet` object with the specified number of buckets and hash function.

**See Also:** `~WCValHashSet`, `bitHash`, `WCExcept::out_of_memory`

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashSet( const WCValHashSet & );
```

**Semantics:** The `WCValHashSet<Type>` is the copy constructor for the `WCValHashSet` class. The new hash is created with the same number of buckets, hash function, all values or pointers stored in the hash, and the exception trap states. If the hash object can be created, but an allocation failure occurs when creating the buckets, the hash will be created with zero buckets. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The `WCValHashSet<Type>` constructor creates a `WCValHashSet` object which is a copy of the passed hash.

**See Also:** `~WCValHashSet`, `operator =`, `WCEXCEPT::out_of_memory`

## ***WCValHashSet<Type>::~~WCValHashSet()***

---

**Synopsis:**

```
#include <wchash.h>
public:
virtual ~WCValHashSet();
```

**Semantics:** The `WCValHashSet<Type>` destructor is the destructor for the `WCValHashSet` class. If the number of elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the hash elements are cleared using the `clear` member function. The call to the `WCValHashSet<Type>` destructor is inserted implicitly by the compiler at the point where the `WCValHashSet` object goes out of scope.

**Results:** The call to the `WCValHashSet<Type>` destructor destroys a `WCValHashSet` object.

**See Also:** `clear`, `WCEXCEPT::not_empty`

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashTable( unsigned (*hash_fn)( const Type & ),
unsigned = WC_DEFAULT_HASH_SIZE );
```

**Semantics:** The `WCValHashTable<Type>` constructor creates a `WCValHashTable` object with no entries and with the number of buckets in the second optional parameter, which defaults to the constant `WC_DEFAULT_HASH_SIZE` (currently defined as 101). The number of buckets specified must be greater than zero, and will be forced to at least one. If the hash object can be created, but an allocation failure occurs when creating the buckets, the table will be created with zero buckets. If the `out_of_memory` exception is enabled, then attempting to insert into a hash table with zero buckets will throw an `out_of_memory` error.

The hash function `hash_fn` is used to determine which bucket each value will be assigned to. If no hash function exists, the static member function `bitHash` is available to help create one.

**Results:** The `WCValHashTable<Type>` constructor creates an initialized `WCValHashTable` object with the specified number of buckets and hash function.

**See Also:** `~WCValHashTable`, `bitHash`, `WCEXCEPT::out_of_memory`

## ***WCValHashTable<Type>::WCValHashTable()***

---

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashTable( unsigned (*hash_fn)( const Type & ),
unsigned = WC_DEFAULT_HASH_SIZE,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the hash. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a hash. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCValHashTableItemSize( Type )
```

**Results:** The `WCValHashTable<Type>` constructor creates an initialized `WCValHashTable` object with the specified number of buckets and hash function.

**See Also:** `~WCValHashTable`, `bitHash`, `WCExcept::out_of_memory`



**Synopsis:**

```
#include <wchash.h>
public:
WCValHashTable( const WCValHashTable & );
```

**Semantics:** The `WCValHashTable<Type>` is the copy constructor for the `WCValHashTable` class. The new hash is created with the same number of buckets, hash function, all values or pointers stored in the hash, and the exception trap states. If the hash object can be created, but an allocation failure occurs when creating the buckets, the hash will be created with zero buckets. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The `WCValHashTable<Type>` constructor creates a `WCValHashTable` object which is a copy of the passed hash.

**See Also:** `~WCValHashTable`, `operator =`, `WCEXCEPT::out_of_memory`

## ***WCValHashTable<Type>::~~WCValHashTable()***

---

**Synopsis:**

```
#include <wchash.h>
public:
virtual ~WCValHashTable();
```

**Semantics:** The `WCValHashTable<Type>` destructor is the destructor for the `WCValHashTable` class. If the number of elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the hash elements are cleared using the `clear` member function. The call to the `WCValHashTable<Type>` destructor is inserted implicitly by the compiler at the point where the `WCValHashTable` object goes out of scope.

**Results:** The call to the `WCValHashTable<Type>` destructor destroys a `WCValHashTable` object.

**See Also:** `clear`, `WCEXCEPT::not_empty`

## *WCValHashTable<Type>::bitHash(), WCValHashSet<Type>::bitHash()*

---

**Synopsis:**

```
#include <wchash.h>
public:
static unsigned bitHash( void *, size_t );
```

**Semantics:** The `bitHash` public member function can be used to implement a hashing function for any type. A hashing value is generated from the value stored for the number of specified bytes pointed to by the first parameter. For example:

```
unsigned my_hash_fn( const int & elem ) {
    return( WCValHashSet<int,String>::bitHash(&elem, sizeof(int));
}
WCValHashSet<int> data_object( &my_hash_fn );
```

**Results:** The `bitHash` public member function returns an unsigned value which can be used as the basis of a user defined hash function.

**See Also:** `WCValHashSet`, `WCValHashTable`

## ***WCValHashTable<Type>::buckets(), WCValHashSet<Type>::buckets()***

---

**Synopsis:**

```
#include <wchash.h>
public:
    unsigned buckets() const;
```

**Semantics:** The `buckets` public member function is used to find the number of buckets contained in the hash object.

**Results:** The `buckets` public member function returns the number of buckets in the hash.

**See Also:** `resize`

**Synopsis:**

```
#include <wchash.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the hash so that it has no entries. The number of buckets remain unaffected. Elements stored in the hash are destroyed using the destructors of `Type`. The hash object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the hash to have no elements.

**See Also:** `~WCValHashSet`, `~WCValHashTable`, `operator =`

## ***WCValHashTable<Type>::contains(), WCValHashSet<Type>::contains()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int contains( const Type & ) const;
```

**Semantics:** The `contains` public member function returns non-zero if the element is stored in the hash, or zero if there is no equivalent element. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The `contains` public member function returns a non-zero value if the element is found in the hash.

**See Also:** `find`

**Synopsis:**

```
#include <wchash.h>
public:
unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to return the current number of elements stored in the hash.

**Results:** The `entries` public member function returns the number of elements in the hash.

**See Also:** `buckets`, `isEmpty`

## ***WCValHashTable<Type>::find(), WCValHashSet<Type>::find()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int find( const Type &, Type & ) const;
```

**Semantics:** The `find` public member function is used to find an element with an equivalent key in the hash. If an equivalent element is found, a non-zero value is returned. The reference to the element passed as the second argument is assigned the found element's value. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The element equivalent to the passed key is located in the hash.



**Synopsis:**

```
#include <wchash.h>
public:
void forAll(
void (*user_fn)( Type, void * ),
void * );
```

**Semantics:** The `forAll` public member function causes the user supplied function to be invoked for every value in the hash. The user function has the prototype

```
void user_func( Type & value, void * data );
```

As the elements are visited, the user function is invoked with the element passed as the first. The second parameter of the `forAll` function is passed as the second parameter to the user function. This value can be used to pass any appropriate data from the main code to the user function.

**Results:** The elements in the hash are all visited, with the user function being invoked for each one.

**See Also:** `find`

## ***WCValHashTable<Type>::insert(), WCValHashSet<Type>::insert()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int insert( const Type & );
```

**Semantics:** The `insert` public member function inserts a value into the hash, using the hash function to determine to which bucket it should be stored. If allocation of the node to store the value fails, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

With a `WCValHashSet`, there must be only one equivalent element in the set. If an element equivalent to the inserted element is already in the hash set, the hash set will remain unchanged, and the `not_unique` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

At some point, the number of buckets initially selected may be too small for the number of elements inserted. The resize of the hash can be controlled by the insertion mechanism by using `WCValHashSet` (or `WCValHashTable`) as a base class, and providing an `insert` member function to do a resize when appropriate. This insert could then call `WCValHashSet::insert` (or `WCValHashTable::insert`) to insert the element. Note that copy constructors and assignment operators are not inherited in your class, but you can provide the following inline definitions (assuming that the class inherited from `WCValHashTable` is named `MyHashTable`):

```
inline MyHashTable( const MyHashTable &orig )
    : WCValHashTable( orig ) {};
inline MyHashTable &operator=( const MyHashTable &orig ) {
    return( WCValHashTable::operator=( orig ) );
}
```

**Results:** The `insert` public member function inserts a value into the hash. If the insert is successful, a non-zero will returned. A zero will be returned if the insert fails.

**See Also:** `operator =`, `WCExcept::out_of_memory`

## ***WCValHashTable<Type>::isEmpty(), WCValHashSet<Type>::isEmpty()***

---

**Synopsis:**

```
#include <wchash.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if the hash is empty.

**Results:** The `isEmpty` public member function returns zero if it contains at least one entry, non-zero if the hash is empty.

**See Also:** `buckets`, `entries`

## ***WCValHashTable<Type>::occurrencesOf()***

---

**Synopsis:**

```
#include <wchash.h>
public:
    unsigned occurrencesOf( const Type & ) const;
```

**Semantics:** The `occurrencesOf` public member function is used to return the current number of elements stored in the hash which are equivalent to the passed value. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The `occurrencesOf` public member function returns the number of elements in the hash.

**See Also:** `buckets`, `entries`, `find`, `isEmpty`

## ***WCValHashTable<Type>::operator =(), WCValHashSet<Type>::operator =()***

---

**Synopsis:**

```
#include <wchash.h>
public:
WCValHashSet & operator =( const WCValHashSet & );
WCValHashTable & operator =( const WCValHashTable & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCValHashTable<Type>` and `WCValHashSet<Type>` classes. The left hand side hash is first cleared using the `clear` member function, and then the right hand side hash is copied. The hash function, exception trap states, and all of the hash elements are copied. If an allocation failure occurs when creating the buckets, the table will be created with zero buckets, and the `out_of_memory` exception is thrown if it is enabled. If there is not enough memory to copy all of the values or pointers in the hash, then only some will be copied, and the `out_of_memory` exception is thrown if it is enabled. The number of entries will correctly reflect the number copied.

**Results:** The `operator =` public member function assigns the left hand side hash to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

## ***WCValHashTable<Type>::operator ==( ), WCValHashSet<Type>::operator ==( )***

---

**Synopsis:**

```
#include <wchash.h>
public:
int operator ==( const WCValHashSet & ) const;
int operator ==( const WCValHashTable & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCValHashTable<Type>` and `WCValHashSet<Type>` classes. Two hash objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side hash are the same object. A FALSE (zero) value is returned otherwise.

## *WCValHashTable<Type>::remove(), WCValHashSet<Type>::remove()*

---

**Synopsis:**

```
#include <wchash.h>
public:
int remove( const Type & );
```

**Semantics:** The `remove` public member function is used to remove the specified element from the hash. If an equivalent element is found, a non-zero value is returned. Zero is returned if the element is not found. If the hash is a table and there is more than one element equivalent to the specified element, then the first equivalent element added to the table is removed. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The element is removed from the hash if it found.

## ***WCValHashTable<Type>::removeAll()***

---

**Synopsis:** `#include <wchash.h>`  
`public:`  
`unsigned removeAll( const Type & );`

**Semantics:** The `removeAll` public member function is used to remove all elements equivalent to the specified element from the hash. Zero is returned if no equivalent elements are found. Note that equivalence is based on the equivalence operator of the element type.

**Results:** All equivalent elements are removed from the hash.



**Synopsis:**

```
#include <wchash.h>
public:
void resize( unsigned );
```

**Semantics:** The `resize` public member function is used to change the number of buckets contained in the hash. If the new number is larger than the previous hash size, then the hash function will be used on all of the stored elements to determine which bucket they should be stored into. Entries are not destroyed or created in the process of being moved. If there is not enough memory to resize the hash, the `out_of_memory` exception is thrown if it is enabled, and the hash will contain the number of buckets it contained before the resize. If the new number is zero, then the `zero_buckets` exception is thrown if it is enabled, and no resize will be performed. The hash is guaranteed to contain the same number of entries after the resize.

**Results:** The hash is resized to the new number of buckets.

**See Also:** `WCEXCEPT::out_of_memory`, `WCEXCEPT::zero_buckets`



---

# 11 Hash Iterators

Hash iterators are used to step through a hash one or more elements at a time. Iterators which are newly constructed or reset are positioned before the first element in the hash. The hash may be traversed one element at a time using the pre-increment or call operator. An increment operation causing the iterator to be positioned after the end of the hash returns zero. Further increments will cause the `undef_iter` exception to be thrown, if it is enabled. The `WCIterExcept` class provides the common exception handling control interface for all of the iterators.

Since the iterator classes are all template classes, most of the functionality was derived from common base classes. In the listing of class member functions, those public member functions which appear to be in the iterator class but are actually defined in the common base class are identified as if they were explicitly specified in the iterator class.

**Declared:** wchiter.h

The WCPtrHashDictIter<Key,Value> class is the templated class used to create iterator objects for WCPtrHashDict<Key,Value> objects. In the description of each member function, the text *Key* is used to indicate the template parameter defining the type of the indices pointed to by the pointers stored in the dictionary. The text *Value* is used to indicate the template parameter defining the type of the data pointed to by the pointers stored in the dictionary. The WCIterExcept class is a base class of the WCPtrHashDictIter<Key,Value> class and provides the exceptions member function. This member function controls the exceptions which can be thrown by the WCPtrHashDictIter<Key,Value> object. No exceptions are enabled unless they are set by the exceptions member function.

### Public Member Functions

The following member functions are declared in the public interface:

```
WCPtrHashDictIter();
WCPtrHashDictIter( const WCPtrHashDict<Key,Value> & );
~WCPtrHashDictIter();
const WCPtrHashDict<Key,Value> *container() const;
Key *key();
void reset();
void reset( WCPtrHashDict<Key,Value> & );
Value * value();
```

### Public Member Operators

The following member operators are declared in the public interface:

```
int operator ()();
int operator ++();
```

**Synopsis:**

```
#include <whiter.h>
public:
WCPtrHashDictIter();
```

**Semantics:** The public `WCPtrHashDictIter<Key,Value>` constructor is the default constructor for the class and initializes the iterator with no hash to operate on. The `reset` member function must be called to provide the iterator with a hash to iterate over.

**Results:** The public `WCPtrHashDictIter<Key,Value>` constructor creates an initialized `WCPtrHashDictIter` hash iterator object.

**See Also:** `~WCPtrHashDictIter`, `reset`

## ***WPtrHashDictIter<Key,Value>::WPtrHashDictIter()***

---

**Synopsis:** `#include <whiter.h>`  
`public:`  
`WPtrHashDictIter( WPtrHashDict<Key,Value> & );`

**Semantics:** The public `WPtrHashDictIter<Key,Value>` constructor is a constructor for the class. The value passed as a parameter is a `WPtrHashDict` hash object. The iterator will be initialized for that hash object and positioned before the first hash element. To position the iterator to a valid element within the hash, increment it using one of the `operator ++` or `operator ()` operators.

**Results:** The public `WPtrHashDictIter<Key,Value>` constructor creates an initialized `WPtrHashDictIter` hash iterator object positioned before the first element in the hash.

**See Also:** `~WPtrHashDictIter`, `operator ()`, `operator ++`, `reset`

**Synopsis:**

```
#include <whiter.h>
public:
~WPtrHashDictIter();
```

**Semantics:** The public `~WPtrHashDictIter<Key, Value>` destructor is the destructor for the class. The call to the destructor is inserted implicitly by the compiler at the point where the `WPtrHashDictIter` hash iterator object goes out of scope.

**Results:** The `WPtrHashDictIter` hash iterator object is destroyed.

**See Also:** `WPtrHashDictIter`

## ***WCPtrHashDictIter<Key,Value>::container()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
WCPtrHashDict<Key,Value> *container() const;
```

**Semantics:** The `container` public member function returns a pointer to the hash container object. If the iterator has not been initialized with a hash object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the hash object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a hash.

**See Also:** `WCPtrHashDictIter`, `reset`, `WCIterExcept::undef_iter`



**Synopsis:**

```
#include <wchiter.h>
public:
Key *key();
```

**Semantics:** The `key` public member function returns a pointer to the `Key` value of the hash item at the current iterator position.

If the iterator is not associated with a hash, or the iterator position is either before the first element or past the last element in the hash, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** A pointer to `Key` at the current iterator element is returned. If the current element is undefined, an undefined pointer is returned.

**See Also:** `operator ()`, `operator ++`, `reset`, `WCIterExcept::undef_item`

## ***WCPtrHashDictIter<Key,Value>::operator ()()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
int operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The hash element which follows the current item is set to be the new current item. If the previous current item was the last element in the hash, the iterator is positioned after the end of the hash.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first hash element, the current item will be set to the first element. If the hash is empty, the iterator will be positioned after the end of the hash.

If the iterator is not associated with a hash or the iterator position before the increment was past the last element the hash, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a non-zero value if the iterator is positioned on a hash item. `Zero(0)` is returned when the iterator is incremented past the end of the hash.

**See Also:** `operator ++`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wchiter.h>
public:
int operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The hash element which follows the current item is set to be the new current item. If the previous current item was the last element in the hash, the iterator is positioned after the end of the hash.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

The current item will be set to the first hash element if the iterator was positioned before the first element in the hash. If the hash is empty, the iterator will be positioned after the end of the hash.

If the iterator is not associated with a hash or the iterator position before the increment was past the last element the hash, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a non-zero value if the iterator is positioned on a hash item. `Zero(0)` is returned when the iterator is incremented past the end of the hash.

**See Also:** `operator ()`, `reset`, `WCIterExcept::undef_iter`

## ***WPtrHashDictIter<Key, Value>::reset()***

---

**Synopsis:**

```
#include <whiter.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated hash.

**Results:** The iterator is positioned before the first hash element.

**See Also:** `WPtrHashDictIter`, `container`

**Synopsis:**

```
#include <wchiter.h>
public:
void reset( WPtrHashDict<Key,Value> & );
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified hash. The iterator is positioned before the first element in the hash.

**Results:** The iterator is positioned before the first element of the specified hash.

**See Also:** `WPtrHashDictIter`, `container`

## ***WCPtrHashDictIter<Key,Value>::value()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
Value *value();
```

**Semantics:** The `value` public member function returns a pointer to the `Value` the current iterator position.

If the iterator is not associated with a hash, or the iterator position is either before the first element or past the last element in the hash, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** A pointer to the `Value` at the current iterator element is returned. If the current element is undefined, an undefined pointer is returned.

**See Also:** `operator ()`, `operator ++`, `reset`, `WCIterExcept::undef_item`

**Declared:** wchiter.h

The WCValHashDictIter<Key,Value> class is the templated class used to create iterator objects for WCValHashDict<Key,Value> objects. In the description of each member function, the text *Key* is used to indicate the template parameter defining the type of the indices used to store data in the dictionary. The text *Value* is used to indicate the template parameter defining the type of the data stored in the dictionary. The WCIterExcept class is a base class of the WCValHashDictIter<Key,Value> class and provides the exceptions member function. This member function controls the exceptions which can be thrown by the WCValHashDictIter<Key,Value> object. No exceptions are enabled unless they are set by the exceptions member function.

### Public Member Functions

The following member functions are declared in the public interface:

```
WCValHashDictIter();
WCValHashDictIter( const WCValHashDict<Key,Value> & );
~WCValHashDictIter();
const WCValHashDict<Key,Value> *container() const;
Key key();
void reset();
void reset( WCValHashDict<Key,Value> & );
Value value();
```

### Public Member Operators

The following member operators are declared in the public interface:

```
int operator ()();
int operator ++();
```

## ***WCValHashDictIter<Key, Value>::WCValHashDictIter()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
    WCValHashDictIter();
```

**Semantics:** The public `WCValHashDictIter<Key, Value>` constructor is the default constructor for the class and initializes the iterator with no hash to operate on. The `reset` member function must be called to provide the iterator with a hash to iterate over.

**Results:** The public `WCValHashDictIter<Key, Value>` constructor creates an initialized `WCValHashDictIter` hash iterator object.

**See Also:** `~WCValHashDictIter`, `reset`



**Synopsis:**

```
#include <wchiter.h>
public:
WCValHashDictIter( WCValHashDict<Key,Value> & );
```

**Semantics:** The public `WCValHashDictIter<Key,Value>` constructor is a constructor for the class. The value passed as a parameter is a `WCValHashDict` hash object. The iterator will be initialized for that hash object and positioned before the first hash element. To position the iterator to a valid element within the hash, increment it using one of the `operator ++` or `operator ()` operators.

**Results:** The public `WCValHashDictIter<Key,Value>` constructor creates an initialized `WCValHashDictIter` hash iterator object positioned before the first element in the hash.

**See Also:** `~WCValHashDictIter`, `operator ()`, `operator ++`, `reset`

## ***WCValHashDictIter<Key, Value>::~~WCValHashDictIter()***

---

**Synopsis:** `#include <wchiter.h>`  
`public:`  
`~WCValHashDictIter();`

**Semantics:** The public `~WCValHashDictIter<Key, Value>` destructor is the destructor for the class. The call to the destructor is inserted implicitly by the compiler at the point where the `WCValHashDictIter` hash iterator object goes out of scope.

**Results:** The `WCValHashDictIter` hash iterator object is destroyed.

**See Also:** `WCValHashDictIter`

**Synopsis:**

```
#include <wchiter.h>
public:
WCVaHashDict<Key,Value> *container() const;
```

**Semantics:** The `container` public member function returns a pointer to the hash container object. If the iterator has not been initialized with a hash object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the hash object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a hash.

**See Also:** `WCVaHashDictIter`, `reset`, `WCIterExcept::undef_iter`

## ***WCValHashDictIter<Key, Value>::key()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
Key key();
```

**Semantics:** The `key` public member function returns the value of `Key` at the current iterator position.

If the iterator is not associated with a hash, or the iterator position is either before the first element or past the last element in the hash, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** The value of `Key` at the current iterator element is returned. If the current element is undefined, a default initialized object is returned.

**See Also:** `operator ()`, `operator ++`, `reset`, `WCIterExcept::undef_item`

**Synopsis:**

```
#include <wchiter.h>
public:
int operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The hash element which follows the current item is set to be the new current item. If the previous current item was the last element in the hash, the iterator is positioned after the end of the hash.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first hash element, the current item will be set to the first element. If the hash is empty, the iterator will be positioned after the end of the hash.

If the iterator is not associated with a hash or the iterator position before the increment was past the last element the hash, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a non-zero value if the iterator is positioned on a hash item. `Zero(0)` is returned when the iterator is incremented past the end of the hash.

**See Also:** `operator ++`, `reset`, `WCIterExcept::undef_iter`

## ***WCValHashDictIter<Key, Value>::operator ++()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
int operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The hash element which follows the current item is set to be the new current item. If the previous current item was the last element in the hash, the iterator is positioned after the end of the hash.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

The current item will be set to the first hash element if the iterator was positioned before the first element in the hash. If the hash is empty, the iterator will be positioned after the end of the hash.

If the iterator is not associated with a hash or the iterator position before the increment was past the last element the hash, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a non-zero value if the iterator is positioned on a hash item. `Zero(0)` is returned when the iterator is incremented past the end of the hash.

**See Also:** `operator ()`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wchiter.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated hash.

**Results:** The iterator is positioned before the first hash element.

**See Also:** `WCVaIHashDictIter`, `container`

## ***WCVaIHashDictIter<Key, Value>::reset()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
void reset( WCVaIHashDict<Key, Value> & );
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified hash. The iterator is positioned before the first element in the hash.

**Results:** The iterator is positioned before the first element of the specified hash.

**See Also:** `WCVaIHashDictIter`, `container`



**Synopsis:**

```
#include <wchiter.h>
public:
Value value();
```

**Semantics:** The `value` public member function returns the value of `Value` at the current iterator position.

If the iterator is not associated with a hash, or the iterator position is either before the first element or past the last element in the hash, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** The value of the `Value` at the current iterator element is returned. If the current element is undefined, a default initialized object is returned.

**See Also:** `operator ()`, `operator ++`, `reset`, `WCIterExcept::undef_item`

## ***WCPtrHashSetIter<Type>, WCPtrHashTableIter<Type>***

---

**Declared:** wchiter.h

The `WCPtrHashSetIter<Type>` and `WCPtrHashTableIter<Type>` classes are the templated classes used to create iterator objects for `WCPtrHashTable<Type>` and `WCPtrHashSet<Type>` objects. In the description of each member function, the text `Type` is used to indicate the hash element type specified as the template parameter. The `WCIterExcept` class is a base class of the `WCPtrHashSetIter<Type>` and `WCPtrHashTableIter<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCPtrHashSetIter<Type>` and `WCPtrHashTableIter<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCPtrHashSetIter();
WCPtrHashSetIter( const WCPtrHashSet<Type> & );
~WCPtrHashSetIter();
WCPtrHashTableIter();
WCPtrHashTableIter( const WCPtrHashTable<Type> & );
~WCPtrHashTableIter();
const WCPtrHashTable<Type> *container() const;
const WCPtrHashSet<Type> *container() const;
Type *current() const;
void reset();
void WCPtrHashSetIter<Type>::reset( WCPtrHashSet<Type> & );
void WCPtrHashTableIter<Type>::reset( WCPtrHashTable<Type> &
);
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
int operator ()();
int operator ++();
```

**Synopsis:**

```
#include <wchiter.h>
public:
WCPtrHashSetIter();
```

**Semantics:** The public `WCPtrHashSetIter<Type>` constructor is the default constructor for the class and initializes the iterator with no hash to operate on. The `reset` member function must be called to provide the iterator with a hash to iterate over.

**Results:** The public `WCPtrHashSetIter<Type>` constructor creates an initialized `WCPtrHashSetIter` hash iterator object.

**See Also:** `~WCPtrHashSetIter`, `WCPtrHashTableIter`, `reset`

## ***WCPtrHashSetIter<Type>::WCPtrHashSetIter()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
WCPtrHashSetIter( WCPtrHashSet<Type> & );
```

**Semantics:** The public `WCPtrHashSetIter<Type>` constructor is a constructor for the class. The value passed as a parameter is a `WCPtrHashSet` hash object. The iterator will be initialized for that hash object and positioned before the first hash element. To position the iterator to a valid element within the hash, increment it using one of the `operator ++` or `operator ()` operators.

**Results:** The public `WCPtrHashSetIter<Type>` constructor creates an initialized `WCPtrHashSetIter` hash iterator object positioned before the first element in the hash.

**See Also:** `~WCPtrHashSetIter`, `operator ()`, `operator ++`, `reset`

**Synopsis:**

```
#include <wchiter.h>
public:
~WPtrHashSetIter();
```

**Semantics:** The public `~WPtrHashSetIter<Type>` destructor is the destructor for the class. The call to the destructor is inserted implicitly by the compiler at the point where the `WPtrHashSetIter` hash iterator object goes out of scope.

**Results:** The `WPtrHashSetIter` hash iterator object is destroyed.

**See Also:** `WPtrHashSetIter`, `WPtrHashTableIter`

## ***WPtrHashTableIter<Type>::WPtrHashTableIter()***

---

**Synopsis:**

```
#include <whiter.h>
public:
WPtrHashTableIter();
```

**Semantics:** The public `WPtrHashTableIter<Type>` constructor is the default constructor for the class and initializes the iterator with no hash to operate on. The `reset` member function must be called to provide the iterator with a hash to iterate over.

**Results:** The public `WPtrHashTableIter<Type>` constructor creates an initialized `WPtrHashTableIter` hash iterator object.

**See Also:** `~WPtrHashTableIter`, `WPtrHashSetIter`, `reset`

**Synopsis:**

```
#include <wchiter.h>
public:
WCPtrHashTableIter( WCPtrHashTable<Type> & );
```

**Semantics:** The public `WCPtrHashTableIter<Type>` constructor is a constructor for the class. The value passed as a parameter is a `WCPtrHashTable` hash object. The iterator will be initialized for that hash object and positioned before the first hash element. To position the iterator to a valid element within the hash, increment it using one of the `operator ++` or `operator ()` operators.

**Results:** The public `WCPtrHashTableIter<Type>` constructor creates an initialized `WCPtrHashTableIter` hash iterator object positioned before the first element in the hash.

**See Also:** `~WCPtrHashTableIter`, `operator ()`, `operator ++`, `reset`

## ***WCPtrHashTableIter<Type>::~~WCPtrHashTableIter()***

---

**Synopsis:** `#include <wchiter.h>`  
`public:`  
`~WCPtrHashTableIter();`

**Semantics:** The `WCPtrHashTableIter<Type>` destructor is the destructor for the class. The call to the destructor is inserted implicitly by the compiler at the point where the `WCPtrHashTableIter` hash iterator object goes out of scope.

**Results:** The `WCPtrHashTableIter` hash iterator object is destroyed.

**See Also:** `WCPtrHashSetIter`, `WCPtrHashTableIter`



**Synopsis:**

```
#include <wchiter.h>
public:
WCPtrHashTable<Type> *WCPtrHashTableIter<Type>::container()
const;
WCPtrHashSet<Type> *WCPtrHashSetIter<Type>::container() const;
```

**Semantics:** The `container` public member function returns a pointer to the hash container object. If the iterator has not been initialized with a hash object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the hash object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a hash.

**See Also:** `WCPtrHashSetIter`, `WCPtrHashTableIter`, `reset`, `WCIterExcept::undef_iter`

## ***WCPtrHashSetIter<Type>::current(), WCPtrHashTableIter<Type>::current()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
Type *current();
```

**Semantics:** The `current` public member function returns a pointer to the hash item at the current iterator position.

If the iterator is not associated with a hash, or the iterator position is either before the first element or past the last element in the hash, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** A pointer to the current iterator element is returned. If the current element is undefined, `NULL(0)` is returned.

**See Also:** `operator ()`, `operator ++`, `reset`, `WCIterExcept::undef_item`

**Synopsis:**

```
#include <wchiter.h>
public:
int operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The hash element which follows the current item is set to be the new current item. If the previous current item was the last element in the hash, the iterator is positioned after the end of the hash.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first hash element, the current item will be set to the first element. If the hash is empty, the iterator will be positioned after the end of the hash.

If the iterator is not associated with a hash or the iterator position before the increment was past the last element the hash, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a non-zero value if the iterator is positioned on a hash item. `Zero(0)` is returned when the iterator is incremented past the end of the hash.

**See Also:** `operator ++`, `reset`, `WCIterExcept::undef_iter`

## ***WCPtrHashSetIter<Type>,WCPtrHashTableIter<Type>::operator ++()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
int operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The hash element which follows the current item is set to be the new current item. If the previous current item was the last element in the hash, the iterator is positioned after the end of the hash.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

The current item will be set to the first hash element if the iterator was positioned before the first element in the hash. If the hash is empty, the iterator will be positioned after the end of the hash.

If the iterator is not associated with a hash or the iterator position before the increment was past the last element the hash, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a non-zero value if the iterator is positioned on a hash item. `Zero(0)` is returned when the iterator is incremented past the end of the hash.

**See Also:** `current`, `operator ()`, `reset`, `WCIterExcept::undef_iter`

## *WCPtrHashSetIter<Type>::reset(), WCPtrHashTableIter<Type>::reset()*

---

**Synopsis:**

```
#include <wchiter.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated hash.

**Results:** The iterator is positioned before the first hash element.

**See Also:** `WCPtrHashSetIter`, `WCPtrHashTableIter`, `container`

## ***WPtrHashSetIter<Type>::reset(), WPtrHashTableIter<Type>::reset()***

---

**Synopsis:**

```
#include <whiter.h>
public:
void WPtrHashSetIter<Type>::reset( WPtrHashSet<Type> & );
void WPtrHashTableIter<Type>::reset( WPtrHashTable<Type> &
);
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified hash. The iterator is positioned before the first element in the hash.

**Results:** The iterator is positioned before the first element of the specified hash.

**See Also:** `WPtrHashSetIter`, `WPtrHashTableIter`, `container`

**Declared:** wchiter.h

The `WCValHashSetIter<Type>` and `WCValHashTableIter<Type>` classes are the templated classes used to create iterator objects for `WCValHashTable<Type>` and `WCValHashSet<Type>` objects. In the description of each member function, the text `Type` is used to indicate the hash element type specified as the template parameter. The `WCIterExcept` class is a base class of the `WCValHashSetIter<Type>` and `WCValHashTableIter<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCValHashSetIter<Type>` and `WCValHashTableIter<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCValHashSetIter();
WCValHashSetIter( const WCValHashSet<Type> & );
~WCValHashSetIter();
WCValHashTableIter();
WCValHashTableIter( const WCValHashTable<Type> & );
~WCValHashTableIter();
const WCValHashTable<Type> *container() const;
const WCValHashSet<Type> *container() const;
Type current() const;
void reset();
void WCValHashSetIter<Type>::reset( WCValHashSet<Type> & );
void WCValHashTableIter<Type>::reset( WCValHashTable<Type> &
);
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
int operator ()();
int operator ++();
```

## ***WCValHashSetIter<Type>::WCValHashSetIter()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
    WCValHashSetIter();
```

**Semantics:** The public `WCValHashSetIter<Type>` constructor is the default constructor for the class and initializes the iterator with no hash to operate on. The `reset` member function must be called to provide the iterator with a hash to iterate over.

**Results:** The public `WCValHashSetIter<Type>` constructor creates an initialized `WCValHashSetIter` hash iterator object.

**See Also:** `~WCValHashSetIter`, `WCValHashTableIter`, `reset`



**Synopsis:**

```
#include <wchiter.h>
public:
WCValHashSetIter( WCValHashSet<Type> & );
```

**Semantics:** The public `WCValHashSetIter<Type>` constructor is a constructor for the class. The value passed as a parameter is a `WCValHashSet` hash object. The iterator will be initialized for that hash object and positioned before the first hash element. To position the iterator to a valid element within the hash, increment it using one of the `operator ++` or `operator ()` operators.

**Results:** The public `WCValHashSetIter<Type>` constructor creates an initialized `WCValHashSetIter` hash iterator object positioned before the first element in the hash.

**See Also:** `~WCValHashSetIter`, `operator ()`, `operator ++`, `reset`

## ***WCValHashSetIter<Type>::~~WCValHashSetIter()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
~WCValHashSetIter();
```

**Semantics:** The public `~WCValHashSetIter<Type>` destructor is the destructor for the class. The call to the destructor is inserted implicitly by the compiler at the point where the `WCValHashSetIter` hash iterator object goes out of scope.

**Results:** The `WCValHashSetIter` hash iterator object is destroyed.

**See Also:** `WCValHashSetIter`, `WCValHashTableIter`

**Synopsis:**

```
#include <whiter.h>
public:
WCValHashTableIter();
```

**Semantics:** The public `WCValHashTableIter<Type>` constructor is the default constructor for the class and initializes the iterator with no hash to operate on. The `reset` member function must be called to provide the iterator with a hash to iterate over.

**Results:** The public `WCValHashTableIter<Type>` constructor creates an initialized `WCValHashTableIter` hash iterator object.

**See Also:** `~WCValHashTableIter`, `WCValHashSetIter`, `reset`

## ***WCValHashTableIter<Type>::WCValHashTableIter()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
    WCValHashTableIter( WCValHashTable<Type> & );
```

**Semantics:** The public `WCValHashTableIter<Type>` constructor is a constructor for the class. The value passed as a parameter is a `WCValHashTable` hash object. The iterator will be initialized for that hash object and positioned before the first hash element. To position the iterator to a valid element within the hash, increment it using one of the `operator ++` or `operator ()` operators.

**Results:** The public `WCValHashTableIter<Type>` constructor creates an initialized `WCValHashTableIter` hash iterator object positioned before the first element in the hash.

**See Also:** `~WCValHashTableIter`, `operator ()`, `operator ++`, `reset`

**Synopsis:**

```
#include <whiter.h>
public:
~WCValHashTableIter();
```

**Semantics:** The `WCValHashTableIter<Type>` destructor is the destructor for the class. The call to the destructor is inserted implicitly by the compiler at the point where the `WCValHashTableIter` hash iterator object goes out of scope.

**Results:** The `WCValHashTableIter` hash iterator object is destroyed.

**See Also:** `WCValHashSetIter`, `WCValHashTableIter`

## ***WCValHashSetIter<Type>,WCValHashTableIter<Type>::container()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
WCValHashTable<Type> *WCValHashTableIter<Type>::container()
const;
WCValHashSet<Type> *WCValHashSetIter<Type>::container() const;
```

**Semantics:** The `container` public member function returns a pointer to the hash container object. If the iterator has not been initialized with a hash object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the hash object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a hash.

**See Also:** `WCValHashSetIter`, `WCValHashTableIter`, `reset`, `WCIterExcept::undef_iter`

## ***WCValHashSetIter<Type>::current(), WCValHashTableIter<Type>::current()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
Type current();
```

**Semantics:** The `current` public member function returns the value of the hash element at the current iterator position.

If the iterator is not associated with a hash, or the iterator position is either before the first element or past the last element in the hash, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** The value at the current iterator element is returned. If the current element is undefined, a default initialized object is returned.

**See Also:** `operator ()`, `operator ++`, `reset`, `WCIterExcept::undef_item`

## *WCValHashSetIter<Type>,WCValHashTableIter<Type>::operator ()()*

---

**Synopsis:**

```
#include <wchiter.h>
public:
int operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The hash element which follows the current item is set to be the new current item. If the previous current item was the last element in the hash, the iterator is positioned after the end of the hash.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first hash element, the current item will be set to the first element. If the hash is empty, the iterator will be positioned after the end of the hash.

If the iterator is not associated with a hash or the iterator position before the increment was past the last element the hash, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a non-zero value if the iterator is positioned on a hash item. `Zero(0)` is returned when the iterator is incremented past the end of the hash.

**See Also:** `operator ++`, `reset`, `WCIterExcept::undef_iter`



**Synopsis:**

```
#include <wchiter.h>
public:
int operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The hash element which follows the current item is set to be the new current item. If the previous current item was the last element in the hash, the iterator is positioned after the end of the hash.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

The current item will be set to the first hash element if the iterator was positioned before the first element in the hash. If the hash is empty, the iterator will be positioned after the end of the hash.

If the iterator is not associated with a hash or the iterator position before the increment was past the last element the hash, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a non-zero value if the iterator is positioned on a hash item. `Zero(0)` is returned when the iterator is incremented past the end of the hash.

**See Also:** `current`, `operator ()`, `reset`, `WCIterExcept::undef_iter`

## ***WCValHashSetIter<Type>::reset(), WCValHashTableIter<Type>::reset()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated hash.

**Results:** The iterator is positioned before the first hash element.

**See Also:** `WCValHashSetIter`, `WCValHashTableIter`, `container`

## ***WCValHashSetIter<Type>::reset(), WCValHashTableIter<Type>::reset()***

---

**Synopsis:**

```
#include <wchiter.h>
public:
void WCValHashSetIter<Type>::reset( WCValHashSet<Type> & );
void WCValHashTableIter<Type>::reset( WCValHashTable<Type> &
);
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified hash. The iterator is positioned before the first element in the hash.

**Results:** The iterator is positioned before the first element of the specified hash.

**See Also:** `WCValHashSetIter`, `WCValHashTableIter`, `container`

***WCValHashSetIter<Type>::reset(), WCValHashTableIter<Type>::reset()***

---

---

# 12 List Containers

List containers are single or double linked lists. The choice of which type of list to use is determined by the direction in which the list is traversed and by what is stored in the list. A list to which items are just added and removed may be most efficiently implemented as a single linked list. If frequent retrievals of items at given indexes within the list are made, double linked lists can offer some improved search performance.

There are three sets of list container classes: value, pointer and intrusive.

Value lists are the simplest to use but have the most requirements on the type stored in the lists. Copies are made of the values stored in the list, which could be undesirable if the stored objects are complicated and copying is expensive. Value lists should not be used to store objects of a base class if any derived types of different sizes would be stored in the list, or if the destructor for the derived class must be called. The `WCValSList<Type>` container class implements single linked value lists, and the `WCValDList<Type>` class double linked value lists.

Pointer list elements store pointers to objects. No creating, copying or destroying of objects stored in the list occurs. The only requirement of the type pointed to is that an equivalence operator is provided so that lookups can be performed. The `WCPtrSList<Type>` class implements single linked pointer lists, and the `WCPtrDList<Type>` class double linked pointer lists.

Intrusive lists require that the list elements are objects derived from the `WCSTLink` or `WCSTLink` class, depending on whether a single or double linked list is used. The list classes require nothing else from the list elements. No creating, destroying or copying of any object is performed by the intrusive list classes, and must be done by the user of the class. One advantage of an intrusive list is a list element can be removed from one list and inserted into another list without creating new list element objects or deleting old objects. The `WCISvSList<Type>` class implements single linked intrusive lists, and the `WCISvDList<Type>` class double linked intrusive lists.

A list may be traversed using the corresponding list iterator class. Iterators allow lists to be stepped through one or more elements at a time. The iterator classes which correspond to single linked list containers have some functionality inhibited. If backward traversal is required, the double linked containers and iterators must be used.

The classes are presented in alphabetical order. The `WCSTLink` and `WCSTLink` class provide a common control interface for the list elements for the intrusive classes.

Since the container classes are all template classes, deriving most of the functionality from common base classes was used. In the listing of class member functions, those public member functions which appear to be in the container class but are actually defined in the common base class are identified as if they were explicitly specified in the container class.

**Declared:** `wclcom.h`

**Derived from:**

`WCSLink`

The `WCDLink` class is the building block for all of the double linked list classes. It is implemented in terms of the `WCSLink` base class. Since no user data is stored directly with it, the `WCDLink` class should only be used as a base class to derive a user defined class.

When creating a double linked intrusive list, the `WCDLink` class is used to derive the user defined class that holds the data to be inserted into the list.

The `wclcom.h` header file is included by the `wclist.h` header file. There is no need to explicitly include the `wclcom.h` header file unless the `wclist.h` header file is not included. No errors will result if it is included.

Note that the destructor is non-virtual so that list elements are of minimum size. Objects created as a class derived from the `WCDLink` class, but destroyed while typed as a `WCDLink` object will not invoke the destructor of the derived class.

**Public Member Functions**

The following public member functions are declared:

```
WCDLink();  
~WCDLink();
```

**See Also:** `WCSLink`

## ***WCDLink::WCDLink()***

---

**Synopsis:**

```
#include <wclist.h>
public:
WCDLink();
```

**Semantics:** The public `WCDLink` constructor creates an `WCDLink` object. The public `WCDLink` constructor is used implicitly by the compiler when it generates a constructor for a derived class.

**Results:** The public `WCDLink` constructor produces an initialized `WCDLink` object.

**See Also:** `~WCDLink`



**Synopsis:** `#include <wclist.h>`  
`public:`  
`~WCDLink();`

**Semantics:** The public `~WCDLink` destructor does not do anything explicit. The call to the public `~WCDLink` destructor is inserted implicitly by the compiler at the point where the object derived from `WCDLink` goes out of scope.

**Results:** The object derived from `WCDLink` is destroyed.

**See Also:** `WCDLink`

## *WCIsvSList<Type>, WCIsvDList<Type>*

---

**Declared:** `wclist.h`

The `WCIsvSList<Type>` and `WCIsvDList<Type>` classes are the templated classes used to create objects which are single or double linked lists. The created list is intrusive, which means that list elements which are inserted must be created with a library supplied base class. The class `WCSSLink` provides the base class definition for single linked lists, and should be inherited by the definition of any list item for single linked lists. It provides the linkage that is used to traverse the list elements. Similarly, the class `WCDDLlink` provides the base class definition for double lists, and should be inherited by the definition of any list item for double lists.

In the description of each member function, the text `Type` is used to indicate the type value specified as the template parameter. `Type` is the type of the list elements, derived from `WCSSLink` or `WCDDLlink`.

The `WCExcept` class is a base class of the `WCIsvSList<Type>` and `WCIsvDList<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCIsvSList<Type>` and `WCIsvDList<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Requirements of Type**

The `WCIsvSList<Type>` class requires only that `Type` is derived from `WCSSLink`. The `WCIsvDList<Type>` class requires only that `Type` is derived from `WCDDLlink`.

### **Private Member Functions**

In an intrusive list, copying a list is undefined. Setting the copy constructor and assignment operator as private is the standard mechanism to ensure a copy cannot be made. The following member functions are declared private:

```
void WCIsvSList( const WCIsvSList & );
void WCIsvDList( const WCIsvDList & );
WCIsvSList & WCIsvSList::operator =( const WCIsvSList & );
WCIsvDList & WCIsvDList::operator =( const WCIsvDList & );
```

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCIsvSList();
~WCIsvSList();
WCIsvDList();
```

```
~WCIsvDList();
int append( Type * );
void clear();
void clearAndDestroy();
int contains( const Type * ) const;
int entries() const;
Type * find( int = 0 ) const;
Type * findLast() const;
void forAll( void (*)( Type *, void * ), void * );
Type * get( int = 0 );
int index( const Type * ) const;
int index( int (*)( const Type *, void * ), void * ) const;
int insert( Type * );
int isEmpty() const;
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
int WCIsvSList::operator ==( const WCIsvSList & ) const;
int WCIsvDList::operator ==( const WCIsvDList & ) const;
```

### **Sample Program Using an Intrusive List**

```
#include <wclist.h>
#include <iostream.h>

class int_ddata : public WCDLink {
public:
    inline int_ddata() {};
    inline int_ddata() {};
    inline int_ddata( int datum ) : info( datum ) {};

    int    info;
};

static void test1( void );

void data_isv_prt( int_ddata * data, void * str ) {
    cout << (char *)str << "[" << data->info << "]\n";
}

void main() {
    try {
        test1();
    } catch( ... ) {
        cout << "we caught an unexpected exception\n";
    }
    cout.flush();
}

void test1 ( void ) {
    WCIsvDList<int_ddata> list;
    int_ddata data1(1);
    int_ddata data2(2);
    int_ddata data3(3);
    int_ddata data4(4);
    int_ddata data5(5);

    list.exceptions( WCEXCEPT::check_all );
    list.append( &data2 );
    list.append( &data3 );
    list.append( &data4 );

    list.insert( &data1 );
    list.append( &data5 );
    cout << "<intrusive double list for int_ddata>\n";
    list.forAll( data_isv_prt, "" );
    data_isv_prt( list.find( 3 ), "<the fourth element>" );
    data_isv_prt( list.get( 2 ), "<the third element>" );
    data_isv_prt( list.get(), "<the first element>" );
    list.clear();
    cout.flush();
}
```

**Synopsis:**

```
#include <wclist.h>
public:
WCIsvSList();
```

**Semantics:** The `WCIsvSList` public member function creates an empty `WCIsvSList` object.

**Results:** The `WCIsvSList` public member function produces an initialized `WCIsvSList` object.

**See Also:** `~WCIsvSList`

## ***WCIsvSList<Type>::WCIsvSList()***

---

**Synopsis:** `#include <wclist.h>`  
`private:`  
`void WCIsvSList( const WCIsvSList & );`

**Semantics:** The `WCIsvSList` private member function is the copy constructor for the single linked list class. Making a copy of the list object would result in a error condition, since intrusive lists cannot share data items with other lists.

**Synopsis:**

```
#include <wclist.h>
public:
~WCIsvSList();
```

**Semantics:** The `~WCIsvSList` public member function destroys the `WCIsvSList` object. If the list is not empty and the `not_empty` exception is enabled, the exception is thrown. If the `not_empty` exception is not enabled and the list is not empty, the list is cleared using the `clear` member function. The call to the `~WCIsvSList` public member function is inserted implicitly by the compiler at the point where the `WCIsvSList` object goes out of scope.

**Results:** The `WCIsvSList` object is destroyed.

**See Also:** `WCIsvSList`, `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

## ***WCIsvDList<Type>::WCIsvDList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
    WCIsvDList();
```

**Semantics:** The `WCIsvDList` public member function creates an empty `WCIsvDList` object.

**Results:** The `WCIsvDList` public member function produces an initialized `WCIsvDList` object.

**See Also:** `~WCIsvDList`



**Synopsis:** `#include <wclist.h>`  
`private:`  
`WCIsvDList( const WCIsvDList & );`

**Semantics:** The `WCIsvDList` private member function is the copy constructor for the double linked list class. Making a copy of the list object would result in a error condition, since intrusive lists cannot share data items with other lists.

## ***WCIsvDList<Type>::~~WCIsvDList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
~WCIsvDList();
```

**Semantics:** The `~WCIsvDList` public member function destroys the `WCIsvDList` object. If the list is not empty and the `not_empty` exception is enabled, the exception is thrown. If the `not_empty` exception is not enabled and the list is not empty, the list is cleared using the `clear` member function. The call to the `~WCIsvDList` public member function is inserted implicitly by the compiler at the point where the `WCIsvDList` object goes out of scope.

**Results:** The `WCIsvDList` object is destroyed.

**See Also:** `WCIsvDList`, `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

**Synopsis:**

```
#include <wclist.h>
public:
int append( Type * );
```

**Semantics:** The `append` public member function is used to append the list element object to the end of the list. The address of (a pointer to) the list element object should be passed, not the value. Since the linkage information is stored in the list element, it is not possible for the element to be in more than one list, or in the same list more than once.

The passed list element should be constructed using the appropriate link class as a base. `WCSLink` must be used as a list element base class for single linked lists, and `WCDLink` must be used as a list element base class for double linked lists.

**Results:** The list element is appended to the end of the list and a `TRUE` value (non-zero) is returned.

**See Also:** `insert`

## *WCIsvSList<Type>::clear(), WCIsvDList<Type>::clear()*

---

**Synopsis:**

```
#include <wclist.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the list object and set it to the state of the object just after the initial construction. The list object is not destroyed and re-created by this operator, so the object destructor is not invoked. The list elements are not cleared. Any list items still in the list are lost unless pointed to by some pointer object in the program code.

If any of the list elements are not allocated with `new` (local variable or global list elements), then the `clear` public member function must be used. When all list elements are allocated with `new`, the `clearAndDestroy` member function should be used.

**Results:** The `clear` public member function resets the list object to the state of the object immediately after the initial construction.

**See Also:** `~WCIsvSList`, `~WCIsvDList`, `clearAndDestroy`, `get`, `operator =`

**Synopsis:**

```
#include <wclist.h>
public:
void clearAndDestroy();
```

**Semantics:** The `clearAndDestroy` public member function is used to clear the list object and set it to the state of the object just after the initial construction. The list object is not destroyed and re-created by this operator, so the object destructor is not invoked. The link elements are deleted before the list is re-initialized.

If any elements in the list were not allocated by the `new` operator, the `clearAndDestroy` public member function must not be called. The `clearAndDestroy` public member function destroys each list element with the destructor for `Type` even if the list element was created as an object derived from `Type`, unless `Type` has a pure virtual destructor.

**Results:** The `clearAndDestroy` public member function resets the list object to the initial state of the object immediately after the initial construction and deletes the list elements.

**See Also:** `clear`, `get`

## ***WClsvSList<Type>::contains(), WClsvDList<Type>::contains()***

---

**Synopsis:**

```
#include <wclist.h>
public:
int contains( const Type * ) const;
```

**Semantics:** The `contains` public member function is used to determine if a list element object is already contained in the list. The address of (a pointer to) the list element object should be passed, not the value. Each list element is compared to the passed element object to determine if it has the same address. Note that the comparison is of the addresses of the elements, not the contained values.

**Results:** Zero(0) is returned if the passed list element object is not found in the list. A non-zero result is returned if the element is found in the list.

**See Also:** `find`, `index`

**Synopsis:**

```
#include <wclist.h>
public:
int entries() const;
```

**Semantics:** The `entries` public member function is used to determine the number of list elements contained in the list object.

**Results:** The number of entries stored in the list is returned, zero(0) is returned if there are no list elements.

**See Also:** `isEmpty`

## *WClsvSList<Type>::find(), WClsvDList<Type>::find()*

---

**Synopsis:**

```
#include <wclist.h>
public:
Type * find( int = 0 ) const;
```

**Semantics:** The `find` public member function returns a pointer to a list element in the list object. The list element is not removed from the list, so care must be taken not to delete the element returned to you. The optional parameter specifies which element to locate, and defaults to the first element. Since the first element of the list is the zero'th element, the last element will be the number of list entries minus one.

If the list is empty and the `empty_container` exception is enabled, the exception is thrown. If the `index_range` exception is enabled, the exception is thrown if the index value is negative or is greater than the number of list entries minus one.

**Results:** A pointer to the selected list element or the closest list element is returned. If the index value is negative, the closest list element is the first element. The last element is the closest element if the index value is greater than the number of list entries minus one. A value of `NULL(0)` is returned if there are no elements in the list.

**See Also:** `findLast`, `get`, `index`, `isEmpty`, `WCEexcept::empty_container`, `WCEexcept::index_range`



**Synopsis:**

```
#include <wclist.h>
public:
Type * findLast() const;
```

**Semantics:** The `findLast` public member function returns a pointer to the last list element in the list object. The list element is not removed from the list, so care must be taken not to delete the element returned to you.

If the list is empty, one of two exceptions can be thrown. If the `empty_container` exception is enabled, it is thrown. The `index_range` exception is thrown if it is enabled and the `empty_container` exception is not enabled.

**Results:** A pointer to the last list element is returned. A value of `NULL(0)` is returned if there are no elements in the list.

**See Also:** `find`, `get`, `isEmpty`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`

## ***WCIsvSList<Type>::forall(), WCIsvDList<Type>::forall()***

---

**Synopsis:** `#include <wclist.h>`  
`public:`  
`void forall( void (*fn)( Type *, void * ), void * );`

**Semantics:** The `forall` public member function is used to cause the function *fn* to be invoked for each list element. The *fn* function should have the prototype

```
void (*fn)( Type *, void * )
```

The first parameter of *fn* shall accept a pointer to the list element currently active. The second argument passed to *fn* is the second argument of the `forall` function. This allows a callback function to be defined which can accept data appropriate for the point at which the `forall` function is invoked.

**See Also:** `WCIsvConstSListIter`, `WCIsvConstDListIter`, `WCIsvSListIter`, `WCIsvDListIter`

**Synopsis:**

```
#include <wclist.h>
public:
Type * get( int = 0 );
```

**Semantics:** The `get` public member function returns a pointer to a list element in the list object. The list element is also removed from the list. The optional parameter specifies which element to remove, and defaults to the first element. Since the first element of the list is the zero'th element, the last element will be the number of list entries minus one.

If the list is empty and the `empty_container` exception is enabled, the exception is thrown. If the `index_range` exception trap is enabled, the exception is thrown if the index value is negative or is greater than the number of list entries minus one.

**Results:** A pointer to the selected list element or the closest list element is removed and returned. If the index value is negative, the closest list element is the first element. The last element is the closest element if the index value is greater than the number of list entries minus one. A value of `NULL(0)` is returned if there are no elements in the list.

**See Also:** `clear`, `clearAndDestroy`, `find`, `index`, `WCEexcept::empty_container`, `WCEexcept::index_range`

## ***WClsvSList<Type>::index(), WClsvDList<Type>::index()***

---

**Synopsis:**

```
#include <wclist.h>
public:
int index( const Type * ) const;
```

**Semantics:** The `index` public member function is used to determine the index of the first list element equivalent to the passed element. The address of (a pointer to) the list element object should be passed, not the value. Each list element is compared to the passed element object to determine if it has the same address. Note that the comparison is of the addresses of the elements, not the contained values.

**Results:** The index of the first element equivalent to the passed element is returned. If the passed element is not in the list, negative one (-1) is returned.

**See Also:** `contains`, `find`, `get`

**Synopsis:**

```
#include <wclist.h>
public:
int index( int (*test_fn)( const Type *, void * ),
void * ) const;
```

**Semantics:** The `index` public member function is used to determine the index of the first list element for which the supplied `test_fn` function returns true. The `test_fn` function must have the prototype:

```
int (*test_fn)( const Type *, void * );
```

Each list element is passed in turn to the `test_fn` function as the first argument. The second parameter passed is the second argument of the `index` function. This allows the `test_fn` callback function to accept data appropriate for the point at which the `index` function is invoked. The supplied `test_fn` shall return a TRUE (non-zero) value when the index of the passed element is desired. Otherwise, a FALSE (zero) value shall be returned.

**Results:** The index of the first list element for which the `test_fn` function returns non-zero is returned. If the `test_fn` function returns zero for all list elements, negative one (-1) is returned.

**See Also:** `contains`, `find`, `get`

## *WClsvSList<Type>::insert(), WClsvDList<Type>::insert()*

---

**Synopsis:**

```
#include <wclist.h>
public:
int insert( Type * );
```

**Semantics:** The `insert` public member function is used to insert the list element object to the beginning of the list. The address of (a pointer to) the list element object should be passed, not the value. Since the linkage information is stored in the list element, it is not possible for the element to be in more than one list, or in the same list more than once.

The passed list element should be constructed using the appropriate link class as a base. `WCSLink` must be used as a list element base class for single linked lists, and `WCDLink` must be used as a list element base class for double linked lists.

**Results:** The list element is inserted as the first element of the list and a `TRUE` value (non-zero) is returned.

**See Also:** `append`

**Synopsis:**

```
#include <wclist.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if a list object has any list elements contained in it.

**Results:** A TRUE value (non-zero) is returned if the list object does not have any list elements contained within it. A FALSE (zero) result is returned if the list contains at least one element.

**See Also:** `entries`

## ***WCIsvSList<Type>::operator =(), WCIsvDList<Type>::operator =()***

---

**Synopsis:** `#include <wclist.h>`  
`private:`  
`WCIsvSList & WCIsvSList::operator =( const WCIsvSList & );`  
`WCIsvDList & WCIsvDList::operator =( const WCIsvDList & );`

**Semantics:** The `operator =` private member function is the assignment operator for the class. Since making a copy of the list object would result in a error condition, it is made inaccessible by making it a private operator.



**Synopsis:**

```
#include <wclist.h>
public:
int WCIsvSList::operator ==( const WCIsvSList & ) const;
int WCIsvDList::operator ==( const WCIsvDList & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCIsvSList<Type>` and `WCIsvDList<Type>` classes. Two list objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side object and the right hand side objects are the same object. A FALSE (zero) value is returned otherwise.

## ***WCPtrSList<Type>, WCPtrDList<Type>***

---

**Declared:** `wclist.h`

The `WCPtrSList<Type>` and `WCPtrDList<Type>` classes are the templated classes used to create objects which are single or double linked lists.

In the description of each member function, the text `Type` is used to indicate the type value specified as the template parameter. The pointers stored in the list point to values of type `Type`.

The `WCEexcept` class is a base class of the `WCPtrSList<Type>` and `WCPtrDList<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCPtrSList<Type>` and `WCPtrDList<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Requirements of Type**

The `WCPtrSList<Type>` and `WCPtrDList<Type>` classes requires `Type` to have:

- (1) an equivalence operator with constant parameters  
`Type::operator ==( const Type & ) const`

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCPtrSList();
WCPtrSList( void * (*)( size_t ), void (*)( void *, size_t ));
WCPtrSList( const WCPtrSList & );
~WCPtrSList();
WCPtrDList();
WCPtrDList( void * (*)( size_t ), void (*)( void *, size_t ));
WCPtrDList( const WCPtrDList & );
~WCPtrDList();
int append( Type * );
void clear();
void clearAndDestroy();
int contains( const Type * ) const;
int entries() const;
Type * find( int = 0 ) const;
Type * findLast() const;
void forAll( void (*)( Type *, void * ), void *) const;
Type * get( int = 0 );
int index( const Type * ) const;
int insert( Type * );
int isEmpty() const;
```

**Public Member Operators**

The following member operators are declared in the public interface:

```
WCPtrSList & WCPtrSList::operator =( const WCPtrSList & );
WCPtrDList & WCPtrDList::operator =( const WCPtrDList & );
int WCPtrSList::operator ==( const WCPtrSList & ) const;
int WCPtrDList::operator ==( const WCPtrDList & ) const;
```

**Sample Program Using a Pointer List**

```
#include <wclist.h>
#include <iostream.h>

static void test1( void );

void data_ptr_prt( int * data, void * str ) {
    cout << (char *)str << "[" << *data << "]\n";
}

void main() {
    try {
        test1();
    } catch( ... ) {
        cout << "we caught an unexpected exception\n";
    }
    cout.flush();
}

void test1 ( void ) {
    WCPtrDList<int>         list;
    int                    data1(1);
    int                    data2(2);
    int                    data3(3);
    int                    data4(4);
    int                    data5(5);

    list.append( &data2 );
    list.append( &data3 );
    list.append( &data4 );

    list.insert( &data1 );
    list.append( &data5 );
    cout << "<pointer double list for int>\n";
    list.forAll( data_ptr_prt, "" );
    data_ptr_prt( list.find( 3 ), "<the fourth element>" );
    data_ptr_prt( list.get( 2 ), "<the third element>" );
    data_ptr_prt( list.get(), "<the first element>" );
    list.clear();
    cout.flush();
}
```

## ***WPtrSList<Type>::WPtrSList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
    WPtrSList();
```

**Semantics:** The `WPtrSList` public member function creates an empty `WPtrSList` object.

**Results:** The `WPtrSList` public member function produces an initialized `WPtrSList` object.

**See Also:** `WPtrSList`, `~WPtrSList`

**Synopsis:**

```
#include <wclist.h>
public:
WCPtrSList( void *(*allocator)( size_t ),
void (*deallocater)( void *, size_t ) );
```

**Semantics:** The `WCPtrSList` public member function creates an empty `WCPtrSList<Type>` object. The *allocator* function is registered to perform all memory allocations of the list elements, and the *deallocater* function to perform all freeing of the list elements' memory. These functions provide the ability to control how the allocation and freeing of memory is performed, allowing for more efficient memory handling than the general purpose global operator `new()` and operator `delete()` can provide. Memory management optimizations may potentially be made through the *allocator* and *deallocater* functions, but are not recommended before managing memory is understood and determined to be worth while.

The *allocator* function shall return a pointer to allocated memory of size at least the argument, or zero(0) if the allocation cannot be performed. Initialization of the memory returned is performed by the `WCPtrSList<Type>` class.

The `WCPtrSList<Type>` class calls the *deallocater* function only on memory allocated by the *allocator* function. The *deallocater* shall free the memory pointed to by the first argument which is of size the second argument. The size passed to the *deallocater* function is guaranteed to be the same size passed to the *allocator* function when the memory was allocated.

The *allocator* and *deallocater* functions may assume that for a list object instance, the *allocator* is always called with the same first argument (the size of the memory to be allocated). The `WCValSListItemSize(Type)` macro returns the size of the elements which are allocated by the *allocator* function.

**Results:** The `WCPtrSList` public member function creates an initialized `WCPtrSList<Type>` object and registers the *allocator* and *deallocater* functions.

**See Also:** `WCPtrSList`, `~WCPtrSList`

## ***WCPtrSList<Type>::WCPtrSList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
void WCPtrSList( const WCPtrSList & );
```

**Semantics:** The `WCPtrSList` public member function is the copy constructor for the single linked list class. All of the list elements are copied to the new list, as well as the exception trap states, and any registered *allocator* and *deallocator* functions.

If all of the elements cannot be copied and the `out_of_memory` is enabled in the list being copied, the exception is thrown. The new list is created in a valid state, even if all of the list elements could not be copied.

**Results:** The `WCPtrSList` public member function produces a copy of the list.

**See Also:** `WCPtrSList`, `~WCPtrSList`, `clear`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wclist.h>
public:
~WPtrSList();
```

**Semantics:** The `~WPtrSList` public member function destroys the `WPtrSList` object. If the list is not empty and the `not_empty` exception is enabled, the exception is thrown. If the `not_empty` exception is not enabled and the list is not empty, the list is cleared using the `clear` member function. The call to the `~WPtrSList` public member function is inserted implicitly by the compiler at the point where the `WPtrSList` object goes out of scope.

**Results:** The `WPtrSList` object is destroyed.

**See Also:** `WPtrSList`, `clear`, `clearAndDestroy`, `WExcept::not_empty`

## ***WPtrDList<Type>::WPtrDList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
WPtrDList();
```

**Semantics:** The `WPtrDList` public member function creates an empty `WPtrDList` object.

**Results:** The `WPtrDList` public member function produces an initialized `WPtrDList` object.

**See Also:** `WPtrDList`, `~WPtrDList`



**Synopsis:**

```
#include <wclist.h>
public:
WCPtrDList( void *(*allocator)( size_t ),
void (*deallocator)( void *, size_t ) );
```

**Semantics:** The `WCPtrDList` public member function creates an empty `WCPtrDList<Type>` object. The *allocator* function is registered to perform all memory allocations of the list elements, and the *deallocator* function to perform all freeing of the list elements' memory. These functions provide the ability to control how the allocation and freeing of memory is performed, allowing for more efficient memory handling than the general purpose global operator `new()` and operator `delete()` can provide. Memory management optimizations may potentially be made through the *allocator* and *deallocator* functions, but are not recommended before managing memory is understood and determined to be worth while.

The *allocator* function shall return a pointer to allocated memory of size at least the argument, or zero(0) if the allocation cannot be performed. Initialization of the memory returned is performed by the `WCPtrDList<Type>` class.

The `WCPtrDList<Type>` class calls the *deallocator* function only on memory allocated by the *allocator* function. The *deallocator* shall free the memory pointed to by the first argument which is of size the second argument. The size passed to the *deallocator* function is guaranteed to be the same size passed to the *allocator* function when the memory was allocated.

The *allocator* and *deallocator* functions may assume that for a list object instance, the *allocator* is always called with the same first argument (the size of the memory to be allocated). The `WCValDListItemSize(Type)` macro returns the size of the elements which are allocated by the *allocator* function.

**Results:** The `WCPtrDList` public member function creates an initialized `WCPtrDList<Type>` object and registers the *allocator* and *deallocator* functions.

**See Also:** `WCPtrDList`, `~WCPtrDList`

## ***WCPtrDList<Type>::WCPtrDList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
WCPtrDList( const WCPtrDList & );
```

**Semantics:** The `WCPtrDList` public member function is the copy constructor for the double linked list class. All of the list elements are copied to the new list, as well as the exception trap states, and any registered *allocator* and *deallocator* functions.

If all of the elements cannot be copied and the `out_of_memory` is enabled in the list being copied, the exception is thrown. The new list is created in a valid state, even if all of the list elements could not be copied.

**Results:** The `WCPtrDList` public member function produces a copy of the list.

**See Also:** `WCPtrDList`, `~WCPtrDList`, `clear`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wlist.h>
public:
~WPtrDLList();
```

**Semantics:** The `~WPtrDLList` public member function destroys the `WPtrDLList` object. If the list is not empty and the `not_empty` exception is enabled, the exception is thrown. If the `not_empty` exception is not enabled and the list is not empty, the list is cleared using the `clear` member function. The call to the `~WPtrDLList` public member function is inserted implicitly by the compiler at the point where the `WPtrDLList` object goes out of scope.

**Results:** The `WPtrDLList` object is destroyed.

**See Also:** `WPtrDLList`, `clear`, `clearAndDestroy`, `WExcept::not_empty`

## ***WCPtrSList<Type>::append(), WCPtrDList<Type>::append()***

---

**Synopsis:**

```
#include <wclist.h>
public:
int append( Type * );
```

**Semantics:** The `append` public member function is used to append the data to the end of the list.

If the `out_of_memory` exception is enabled and the `append` fails, the exception is thrown.

**Results:** The data element is appended to the end of the list. A `TRUE` value (non-zero) is returned if the `append` is successful. A `FALSE` (zero) result is returned if the `append` fails.

**See Also:** `insert`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wclist.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the list object and set it to the state of the object just after the initial construction. The list object is not destroyed and re-created by this operator, so the object destructor is not invoked.

**Results:** The `clear` public member function resets the list object to the state of the object immediately after the initial construction.

**See Also:** `~WPtrSList`, `~WPtrDList`, `clearAndDestroy`, `get`, `operator =`

## ***WCPtrSList<Type>,WCPtrDList<Type>::clearAndDestroy()***

---

**Synopsis:**

```
#include <wclist.h>
public:
void clearAndDestroy();
```

**Semantics:** The `clearAndDestroy` public member function is used to clear the list object and set it to the state of the object just after the initial construction. The list object is not destroyed and re-created by this operator, so the object destructor is not invoked. Before the list object is re-initialized, the the values pointed to by the list elements are deleted.

**Results:** The `clearAndDestroy` public member function resets the list object to the initial state of the object immediately after the initial construction and deletes the list elements.

**See Also:** `clear`, `get`

**Synopsis:**

```
#include <wclist.h>
public:
int contains( const Type * ) const;
```

**Semantics:** The `contains` public member function is used to determine if a list element object is already contained in the list. Each list element is compared to the passed element using `Type`'s `operator ==` to determine if the passed element is contained in the list. Note that the comparison is of the objects pointed to.

**Results:** Zero(0) is returned if the passed list element object is not found in the list. A non-zero result is returned if the element is found in the list.

**See Also:** `find`, `index`

## ***WPtrSList<Type>::entries(), WPtrDList<Type>::entries()***

---

**Synopsis:**

```
#include <wclist.h>
public:
int entries() const;
```

**Semantics:** The `entries` public member function is used to determine the number of list elements contained in the list object.

**Results:** The number of entries stored in the list is returned, zero(0) is returned if there are no list elements.

**See Also:** `isEmpty`



**Synopsis:**

```
#include <wclist.h>
public:
Type * find( int = 0 ) const;
```

**Semantics:** The `find` public member function returns the value of a list element in the list object. The optional parameter specifies which element to locate, and defaults to the first element. Since the first element of the list is the zero'th element, the last element will be the number of list entries minus one.

If the list is empty and the `empty_container` exception is enabled, the exception is thrown. If the `index_range` exception is enabled, the exception is thrown if the index value is negative or is greater than the number of list entries minus one.

**Results:** The value of the selected list element or the closest element is returned. If the index value is negative, the closest list element is the first element. The last element is the closest element if the index value is greater than the number of list entries minus one. An uninitialized pointer is returned if there are no elements in the list.

**See Also:** `findLast`, `get`, `index`, `isEmpty`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`

## ***WPtrSList<Type>::findLast(), WPtrDList<Type>::findLast()***

---

**Synopsis:**

```
#include <wclist.h>
public:
Type * findLast() const;
```

**Semantics:** The `findLast` public member function returns the value of the last list element in the list object.

If the list is empty, one of two exceptions can be thrown. If the `empty_container` exception is enabled, it is thrown. The `index_range` exception is thrown if it is enabled and the `empty_container` exception is not enabled.

**Results:** The value of the last list element is returned. An uninitialized pointer is returned if there are no elements in the list.

**See Also:** `find`, `get`, `isEmpty`, `WExcept::empty_container`, `WExcept::index_range`

**Synopsis:**

```
#include <wclist.h>
public:
void forall( void (*)( Type *, void * ), void *) const;
```

**Semantics:** The `forall` public member function is used to cause the function *fn* to be invoked for each list element. The *fn* function should have the prototype

```
void (*fn)( Type *, void * )
```

The first parameter of *fn* shall accept the value of the list element currently active. The second argument passed to *fn* is the second argument of the `forall` function. This allows a callback function to be defined which can accept data appropriate for the point at which the `forall` function is invoked.

**See Also:** `WCPtrConstSListIter`, `WCPtrConstDListIter`, `WCPtrSListIter`, `WCPtrDListIter`

## ***WCPtrSList<Type>::get(), WCPtrDList<Type>::get()***

---

**Synopsis:**

```
#include <wclist.h>
public:
Type * get( int = 0 );
```

**Semantics:** The `get` public member function returns the value of the list element in the list object. The list element is also removed from the list. The optional parameter specifies which element to remove, and defaults to the first element. Since the first element of the list is the zero'th element, the last element will be the number of list entries minus one.

If the list is empty and the `empty_container` exception is enabled, the exception is thrown. If the `index_range` exception trap is enabled, the exception is thrown if the index value is negative or is greater than the number of list entries minus one.

**Results:** The value of the selected list element or the closest element is removed and returned. If the index value is negative, the closest list element is the first element. The last element is the closest element if the index value is greater than the number of list entries minus one. An uninitialized pointer is returned if there are no elements in the list.

**See Also:** `clear`, `clearAndDestroy`, `find`, `index`, `WCEexcept::empty_container`, `WCEexcept::index_range`

**Synopsis:**

```
#include <wclist.h>
public:
int index( const Type * ) const;
```

**Semantics:** The `index` public member function is used to determine the index of the first list element equivalent to the passed element. Each list element is compared to the passed element using `Type`'s `operator ==` until the passed element is found, or all list elements have been checked. Note that the comparison is of the objects pointed to.

**Results:** The index of the first element equivalent to the passed element is returned. If the passed element is not in the list, negative one (-1) is returned.

**See Also:** `contains`, `find`, `get`

## ***WCPtrSList<Type>::insert(), WCPtrDList<Type>::insert()***

---

**Synopsis:**

```
#include <wclist.h>
public:
int insert( Type * );
```

**Semantics:** The `insert` public member function is used to insert the data as the first element of the list.

If the `out_of_memory` exception is enabled and the insert fails, the exception is thrown.

**Results:** The data element is inserted into the beginning of the list. A `TRUE` value (non-zero) is returned if the insert is successful. A `FALSE` (zero) result is returned if the insert fails.

**See Also:** `append`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wclist.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if a list object has any list elements contained in it.

**Results:** A TRUE value (non-zero) is returned if the list object does not have any list elements contained within it. A FALSE (zero) result is returned if the list contains at least one element.

**See Also:** `entries`

## ***WCPtrSList<Type>::operator =(), WCPtrDList<Type>::operator =()***

---

**Synopsis:**

```
#include <wclist.h>
public:
WCPtrSList & WCPtrSList::operator =( const WCPtrSList & );
WCPtrDList & WCPtrDList::operator =( const WCPtrDList & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the class. The left hand side of the assignment is first cleared with the `clear` member function. All elements in the right hand side list are then copied, as well as the exception trap states, and any registered *allocator* and *deallocator* functions.

If all of the elements cannot be copied and the `out_of_memory` is enabled in the right hand side list, the exception is thrown. The new list is created in a valid state, even if all of the list elements could not be copied.

**Results:** The `operator =` public member function assigns the right hand side to the left hand side and returns a reference to the left hand side.

**See Also:** `WCPtrSList`, `WCPtrDList`, `clear`, `WCEXCEPT::out_of_memory`



**Synopsis:**

```
#include <wclist.h>
public:
int WCPtrSList::operator ==( const WCPtrSList & ) const;
int WCPtrDList::operator ==( const WCPtrDList & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCPtrSList<Type>` and `WCPtrDList<Type>` classes. Two list objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side object and the right hand side objects are the same object. A FALSE (zero) value is returned otherwise.

**Declared:** `wclcom.h`

**Derived by:** `WCDSLink`

The `WCSLink` class is the building block for all of the list classes. It provides the link that is used to traverse the list elements. The double link classes use the `WCSLink` class to implement both links. Since no user data is stored directly with it, the `WCSLink` class should only be used as a base class to derive a user defined class.

When creating a single linked intrusive list, the `WCSLink` class is used to derive the user defined class that holds the data to be inserted into the list.

The `wclcom.h` header file is included by the `wclist.h` header file. There is no need to explicitly include the `wclcom.h` header file unless the `wclist.h` header file is not included. No errors will result if it is included unnecessarily.

Note that the destructor is non-virtual so that list elements are of minimum size. Objects created as a class derived from the `WCSLink` class, but destroyed while typed as a `WCSLink` object will not invoke the destructor of the derived class.

### Public Member Functions

The following public member functions are declared:

```
WCSLink();  
~WCSLink();
```

**See Also:** `WCDSLink`

**Synopsis:**

```
#include <wclcom.h>
public:
WCSLink();
```

**Semantics:** The public `WCSLink` constructor creates an `WCSLink` object. The public `WCSLink` constructor is used implicitly by the compiler when it generates a constructor for a derived class.

**Results:** The public `WCSLink` constructor produces an initialized `WCSLink` object.

**See Also:** `~WCSLink`

## ***WCSLink::~WCSLink()***

---

**Synopsis:** `#include <wclcom.h>`  
`public:`  
`~WCSLink();`

**Semantics:** The public `~WCSLink` destructor does not do anything explicit. The call to the public `~WCSLink` destructor is inserted implicitly by the compiler at the point where the object derived from `WCSLink` goes out of scope.

**Results:** The object derived from `WCSLink` is destroyed.

**See Also:** `WCSLink`

**Declared:** wclist.h

The WCValSList<Type> and WCValDList<Type> classes are the templated classes used to create objects which are single or double linked lists. Values are copied into the list, which could be undesirable if the stored objects are complicated and copying is expensive. Value lists should not be used to store objects of a base class if any derived types of different sizes would be stored in the list, or if the destructor for a derived class must be called.

In the description of each member function, the text `Type` is used to indicate the type value specified as the template parameter. `Type` is the type of the values stored in the list.

The WCEXCEPT class is a base class of the WCValSList<Type> and WCValDList<Type> classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the WCValSList<Type> and WCValDList<Type> objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### Requirements of Type

The WCValSList<Type> and WCValDList<Type> classes requires `Type` to have:

- (1) a default constructor (`Type::Type()`).
- (2) a well defined copy constructor (`Type::Type( const Type & )`).
- (3) an equivalence operator with constant parameters  
`Type::operator ==( const Type & ) const`

### Public Member Functions

The following member functions are declared in the public interface:

```
WCValSList();
WCValSList( void * (*)( size_t ), void (*)( void *, size_t ) );
WCValSList( const WCValSList & );
~WCValSList();
WCValDList();
WCValDList( void * (*)( size_t ), void (*)( void *, size_t ) );
WCValDList( const WCValDList & );
~WCValDList();
int append( const Type & );
void clear();
void clearAndDestroy();
int contains( const Type & ) const;
int entries() const;
```

## ***WCValSList<Type>, WCValDList<Type>***

---

```
Type find( int = 0 ) const;
Type findLast() const;
void forAll( void (*)( Type, void * ), void *) const;
Type get( int = 0 );
int index( const Type & ) const;
int insert( const Type & );
int isEmpty() const;
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
WCValSList & WCValSList::operator =( const WCValSList & );
WCValDList & WCValDList::operator =( const WCValDList & );
int WCValSList::operator ==( const WCValSList & ) const;
int WCValDList::operator ==( const WCValDList & ) const;
```

### **Sample Program Using a Value List**

```
#include <wclist.h>
#include <iostream.h>

static void test1( void );

void data_val_prt( int data, void * str ) {
    cout << (char *)str << "[" << data << "]"<< "\n";
}

void main() {
    try {
        test1();
    } catch( ... ) {
        cout << "we caught an unexpected exception\n";
    }
    cout.flush();
}

void test1 ( void ) {
    WCValDList<int> list;

    list.append( 2 );
    list.append( 3 );
    list.append( 4 );

    list.insert( 1 );
    list.append( 5 );
    cout << "<value double list for int>\n";
    list.forAll( data_val_prt, "" );
    data_val_prt( list.find( 3 ), "<the fourth element>" );
    data_val_prt( list.get( 2 ), "<the third element>" );
    data_val_prt( list.get(), "<the first element>" );
    list.clear();
    cout.flush();
}
```

## ***WCValsList<Type>::WCValsList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
WCValsList();
```

**Semantics:** The `WCValsList` public member function creates an empty `WCValsList` object.

**Results:** The `WCValsList` public member function produces an initialized `WCValsList` object.

**See Also:** `WCValsList`, `~WCValsList`



**Synopsis:**

```
#include <wclist.h>
public:
WCValSList( void *(*allocator)( size_t ),
void (*deallocator)( void *, size_t ) );
```

**Semantics:** The `WCValSList` public member function creates an empty `WCValSList<Type>` object. The *allocator* function is registered to perform all memory allocations of the list elements, and the *deallocator* function to perform all freeing of the list elements' memory. These functions provide the ability to control how the allocation and freeing of memory is performed, allowing for more efficient memory handling than the general purpose global operator `new()` and operator `delete()` can provide. Memory management optimizations may potentially be made through the *allocator* and *deallocator* functions, but are not recommended before managing memory is understood and determined to be worth while.

The *allocator* function shall return a pointer to allocated memory of size at least the argument, or zero(0) if the allocation cannot be performed. Initialization of the memory returned is performed by the `WCValSList<Type>` class.

The `WCValSList<Type>` class calls the *deallocator* function only on memory allocated by the *allocator* function. The *deallocator* shall free the memory pointed to by the first argument which is of size the second argument. The size passed to the *deallocator* function is guaranteed to be the same size passed to the *allocator* function when the memory was allocated.

The *allocator* and *deallocator* functions may assume that for a list object instance, the *allocator* is always called with the same first argument (the size of the memory to be allocated). The `WCValSListItemSize(Type)` macro returns the size of the elements which are allocated by the *allocator* function.

**Results:** The `WCValSList` public member function creates an initialized `WCValSList<Type>` object and registers the *allocator* and *deallocator* functions.

**See Also:** `WCValSList`, `~WCValSList`

## ***WCValSList<Type>::WCValSList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
void WCValSList( const WCValSList & );
```

**Semantics:** The `WCValSList` public member function is the copy constructor for the single linked list class. All of the list elements are copied to the new list, as well as the exception trap states, and any registered *allocator* and *deallocator* functions. `Type`'s copy constructor is invoked to copy the values contained by the list elements.

If all of the elements cannot be copied and the `out_of_memory` is enabled in the list being copied, the exception is thrown. The new list is created in a valid state, even if all of the list elements could not be copied.

**Results:** The `WCValSList` public member function produces a copy of the list.

**See Also:** `WCValSList`, `~WCValSList`, `clear`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wclist.h>
public:
~WCValSList();
```

**Semantics:** The `~WCValSList` public member function destroys the `WCValSList` object. If the list is not empty and the `not_empty` exception is enabled, the exception is thrown. If the `not_empty` exception is not enabled and the list is not empty, the list is cleared using the `clear` member function. The call to the `~WCValSList` public member function is inserted implicitly by the compiler at the point where the `WCValSList` object goes out of scope.

**Results:** The `WCValSList` object is destroyed.

**See Also:** `WCValSList`, `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

## ***WCValDList<Type>::WCValDList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
    WCValDList();
```

**Semantics:** The `WCValDList` public member function creates an empty `WCValDList` object.

**Results:** The `WCValDList` public member function produces an initialized `WCValDList` object.

**See Also:** `WCValDList`, `~WCValDList`

**Synopsis:**

```
#include <wclist.h>
public:
WCValDList( void *(*allocator)( size_t ),
void (*deallocater)( void *, size_t ) );
```

**Semantics:** The WCValDList public member function creates an empty WCValDList<Type> object. The *allocator* function is registered to perform all memory allocations of the list elements, and the *deallocater* function to perform all freeing of the list elements' memory. These functions provide the ability to control how the allocation and freeing of memory is performed, allowing for more efficient memory handling than the general purpose global operator `new()` and operator `delete()` can provide. Memory management optimizations may potentially be made through the *allocator* and *deallocater* functions, but are not recommended before managing memory is understood and determined to be worth while.

The *allocator* function shall return a pointer to allocated memory of size at least the argument, or zero(0) if the allocation cannot be performed. Initialization of the memory returned is performed by the WCValDList<Type> class.

The WCValDList<Type> class calls the *deallocater* function only on memory allocated by the *allocator* function. The *deallocater* shall free the memory pointed to by the first argument which is of size the second argument. The size passed to the *deallocater* function is guaranteed to be the same size passed to the *allocator* function when the memory was allocated.

The *allocator* and *deallocater* functions may assume that for a list object instance, the *allocator* is always called with the same first argument (the size of the memory to be allocated). The WCValDListItemSize(Type) macro returns the size of the elements which are allocated by the *allocator* function.

**Results:** The WCValDList public member function creates an initialized WCValDList<Type> object and registers the *allocator* and *deallocater* functions.

**See Also:** WCValDList, ~WCValDList

## ***WCValDList<Type>::WCValDList()***

---

**Synopsis:**

```
#include <wclist.h>
public:
WCValDList( const WCValDList & );
```

**Semantics:** The `WCValDList` public member function is the copy constructor for the double linked list class. All of the list elements are copied to the new list, as well as the exception trap states, and any registered *allocator* and *deallocator* functions. `Type`'s copy constructor is invoked to copy the values contained by the list elements.

If all of the elements cannot be copied and the `out_of_memory` is enabled in the list being copied, the exception is thrown. The new list is created in a valid state, even if all of the list elements could not be copied.

**Results:** The `WCValDList` public member function produces a copy of the list.

**See Also:** `WCValDList`, `~WCValDList`, `clear`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wclist.h>
public:
~WCValDList();
```

**Semantics:** The `~WCValDList` public member function destroys the `WCValDList` object. If the list is not empty and the `not_empty` exception is enabled, the exception is thrown. If the `not_empty` exception is not enabled and the list is not empty, the list is cleared using the `clear` member function. The call to the `~WCValDList` public member function is inserted implicitly by the compiler at the point where the `WCValDList` object goes out of scope.

**Results:** The `WCValDList` object is destroyed.

**See Also:** `WCValDList`, `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

## ***WCValSList<Type>::append(), WCValDList<Type>::append()***

---

**Synopsis:**

```
#include <wclist.h>
public:
int append( const Type & );
```

**Semantics:** The `append` public member function is used to append the data to the end of the list. The data stored in the list is a copy of the data passed as a parameter.

If the `out_of_memory` exception is enabled and the `append` fails, the exception is thrown.

**Results:** The data element is appended to the end of the list. A `TRUE` value (non-zero) is returned if the `append` is successful. A `FALSE` (zero) result is returned if the `append` fails.

**See Also:** `insert`, `WCExcept::out_of_memory`



**Synopsis:**

```
#include <wclist.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the list object and set it to the state of the object just after the initial construction. The list object is not destroyed and re-created by this operator, so the object destructor is not invoked.

The `clear` public member function has the same semantics as the `clearAndDestroy` member function.

**Results:** The `clear` public member function resets the list object to the state of the object immediately after the initial construction.

**See Also:** `~WCValSList`, `~WCValDList`, `clearAndDestroy`, `get`, `operator =`

## ***WCValSList<Type>,WCValDList<Type>::clearAndDestroy()***

---

**Synopsis:**

```
#include <wclist.h>
public:
void clearAndDestroy();
```

**Semantics:** The `clearAndDestroy` public member function is used to clear the list object and set it to the state of the object just after the initial construction. The list object is not destroyed and re-created by this operator, so the object destructor is not invoked.

Before the list object is re-initialized, the delete operator is called for each list element.

**Results:** The `clearAndDestroy` public member function resets the list object to the initial state of the object immediately after the initial construction.

**See Also:** `clear`, `get`

**Synopsis:**

```
#include <wclist.h>
public:
int contains( const Type & ) const;
```

**Semantics:** The `contains` public member function is used to determine if a list element object is already contained in the list. Each list element is compared to the passed element using `Type`'s `operator ==` to determine if the passed element is contained in the list.

**Results:** Zero(0) is returned if the passed list element object is not found in the list. A non-zero result is returned if the element is found in the list.

**See Also:** `find`, `index`

## ***WCValSList<Type>::entries(), WCValDList<Type>::entries()***

---

**Synopsis:**

```
#include <wclist.h>
public:
int entries() const;
```

**Semantics:** The `entries` public member function is used to determine the number of list elements contained in the list object.

**Results:** The number of entries stored in the list is returned, zero(0) is returned if there are no list elements.

**See Also:** `isEmpty`

**Synopsis:**

```
#include <wclist.h>
public:
Type find( int = 0 ) const;
```

**Semantics:** The `find` public member function returns the value of a list element in the list object. The optional parameter specifies which element to locate, and defaults to the first element. Since the first element of the list is the zero'th element, the last element will be the number of list entries minus one.

If the list is empty and the `empty_container` exception is enabled, the exception is thrown. If the `index_range` exception is enabled, the exception is thrown if the index value is negative or is greater than the number of list entries minus one.

**Results:** The value of the selected list element or the closest element is returned. If the index value is negative, the closest list element is the first element. The last element is the closest element if the index value is greater than the number of list entries minus one. A default initialized value is returned if there are no elements in the list.

**See Also:** `findLast`, `get`, `index`, `isEmpty`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`

## ***WCValSList<Type>::findLast(), WCValDList<Type>::findLast()***

---

**Synopsis:**

```
#include <wclist.h>
public:
Type findLast() const;
```

**Semantics:** The `findLast` public member function returns the value of the last list element in the list object.

If the list is empty, one of two exceptions can be thrown. If the `empty_container` exception is enabled, it is thrown. The `index_range` exception is thrown if it is enabled and the `empty_container` exception is not enabled.

**Results:** The value of the last list element is returned. A default initialized value is returned if there are no elements in the list.

**See Also:** `find`, `get`, `isEmpty`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`

**Synopsis:**

```
#include <wclist.h>
public:
void forAll( void (*)( Type, void * ), void *) const;
```

**Semantics:** The `forAll` public member function is used to cause the function *fn* to be invoked for each list element. The *fn* function should have the prototype

```
void (*fn)( Type, void * )
```

The first parameter of *fn* shall accept the value of the list element currently active. The second argument passed to *fn* is the second argument of the `forAll` function. This allows a callback function to be defined which can accept data appropriate for the point at which the `forAll` function is invoked.

**See Also:** `WCValConstSListIter`, `WCValConstDListIter`, `WCValSListIter`, `WCValDListIter`

## ***WCValSList<Type>::get(), WCValDList<Type>::get()***

---

**Synopsis:**

```
#include <wclist.h>
public:
Type get( int = 0 );
```

**Semantics:** The `get` public member function returns the value of the list element in the list object. The list element is also removed from the list. The optional parameter specifies which element to remove, and defaults to the first element. Since the first element of the list is the zero'th element, the last element will be the number of list entries minus one.

If the list is empty and the `empty_container` exception is enabled, the exception is thrown. If the `index_range` exception trap is enabled, the exception is thrown if the index value is negative or is greater than the number of list entries minus one.

**Results:** The value of the selected list element or the closest element is removed and returned. If the index value is negative, the closest list element is the first element. The last element is the closest element if the index value is greater than the number of list entries minus one. A default initialized value is returned if there are no elements in the list.

**See Also:** `clear`, `clearAndDestroy`, `find`, `index`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`



**Synopsis:**

```
#include <wclist.h>
public:
int index( const Type & ) const;
```

**Semantics:** The `index` public member function is used to determine the index of the first list element equivalent to the passed element. Each list element is compared to the passed element using `Type`'s `operator ==` until the passed element is found, or all list elements have been checked.

**Results:** The index of the first element equivalent to the passed element is returned. If the passed element is not in the list, negative one (-1) is returned.

**See Also:** `contains`, `find`, `get`

## ***WCVaISList<Type>::insert(), WCVaIDList<Type>::insert()***

---

**Synopsis:**

```
#include <wclist.h>
public:
int insert( const Type & );
```

**Semantics:** The `insert` public member function is used to insert the data as the first element of the list. The data stored in the list is a copy of the data passed as a parameter.

If the `out_of_memory` exception is enabled and the insert fails, the exception is thrown.

**Results:** The data element is inserted into the beginning of the list. A `TRUE` value (non-zero) is returned if the insert is successful. A `FALSE` (zero) result is returned if the insert fails.

**See Also:** `append`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wclist.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if a list object has any list elements contained in it.

**Results:** A TRUE value (non-zero) is returned if the list object does not have any list elements contained within it. A FALSE (zero) result is returned if the list contains at least one element.

**See Also:** `entries`

## ***WCValSList<Type>::operator =(), WCValDList<Type>::operator =()***

---

**Synopsis:**

```
#include <wclist.h>
public:
WCValSList & WCValSList::operator =( const WCValSList & );
WCValDList & WCValDList::operator =( const WCValDList & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the class. The left hand side of the assignment is first cleared with the `clear` member function. All elements in the right hand side list are then copied, as well as the exception trap states, and any registered *allocator* and *deallocator* functions. `Type`'s copy constructor is invoked to copy the values contained by the list elements.

If all of the elements cannot be copied and the `out_of_memory` is enabled in the right hand side list, the exception is thrown. The new list is created in a valid state, even if all of the list elements could not be copied.

**Results:** The `operator =` public member function assigns the right hand side to the left hand side and returns a reference to the left hand side.

**See Also:** `WCValSList`, `WCValDList`, `clear`, `WCEXCEPT::out_of_memory`

## ***WCValSList<Type>::operator ==( ), WCValDList<Type>::operator ==( )***

---

**Synopsis:** `#include <wclist.h>`  
`public:`  
`int WCValSList::operator ==( const WCValSList & ) const;`  
`int WCValDList::operator ==( const WCValDList & ) const;`

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCValSList<Type>` and `WCValDList<Type>` classes. Two list objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side object and the right hand side objects are the same object. A FALSE (zero) value is returned otherwise.



---

# 13 List Iterators

List iterators operate on single or double linked lists. They are used to step through a list one or more elements at a time. The choice of which type of iterator to use is determined by the list you wish to iterate over. For example, to iterate over a non-constant `WCISvDList<Type>` object, use the `WCISvDListIter<Type>` class. A constant `WCValSList<Type>` object can be iterated using the `WCValConstSListIter<Type>` class. The iterators which correspond to the single link list containers have some functionality inhibited. If backward traversal is required, the double linked containers and corresponding iterators must be used.

Like all WATCOM iterators, newly constructed and reset iterators are positioned before the first element in the list. The list may be traversed one element at a time using the pre-increment or call operator. An increment operation causing the iterator to be positioned after the end of the list returns zero. Further increments will cause the `undef_iter` exception to be thrown, if it is enabled. This behaviour allows lists to be traversed simply using a while loop, and is demonstrated in the examples for the iterator classes.

The classes are presented in alphabetical order. The `WCIterExcept` class provides the common exception handling control interface for all of the iterators.

Since the iterator classes are all template classes, deriving most of the functionality from common base classes was used. In the listing of class member functions, those public member functions which appear to be in the iterator class but are actually defined in the common base class are identified as if they were explicitly specified in the iterator class.

**Declared:** wclistit.h

The `WCIsvConstSListIter<Type>` and `WCIsvConstDListIter<Type>` classes are the templated classes used to create iterator objects for constant single and double linked list objects. These classes may be used to iterate over non-constant lists, but the `WCIsvDListIter<Type>` and `WCIsvSListIter<Type>` classes provide additional functionality for only non-constant lists.

In the description of each member function, the text `Type` is used to indicate the list element type specified as the template parameter.

The `WCIterExcept` class is a base class of the `WCIsvConstSListIter<Type>` and `WCIsvConstDListIter<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCIsvConstSListIter<Type>` and `WCIsvConstDListIter<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Private Member Functions**

Some functionality supported by base classes of the iterator are not appropriate for the constant list iterator classes. Setting those functions as private members in the derived class is the standard mechanism to prevent them from being invoked.

```
int append( Type * );
int insert( Type * );
```

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCIsvConstSListIter();
WCIsvConstSListIter( const WCIsvSList<Type> & );
~WCIsvConstSListIter();
WCIsvConstDListIter();
WCIsvConstDListIter( const WCIsvDList<Type> & );
~WCIsvConstDListIter();
const WCIsvSList<Type> *WCIsvConstSListIter<Type>::container()
const;
const WCIsvDList<Type> *WCIsvConstDListIter<Type>::container()
const;
Type * current() const;
void reset();
void WCIsvConstSListIter<Type>::reset( const WCIsvSList<Type>
& );
```



```
void WCIsvConstDListIter<Type>::reset( const WCIsvDList<Type>
& );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
Type * operator ()();
Type * operator ++();
Type * operator +=( int );
```

In the iterators for double linked lists only:

```
Type * operator --();
Type * operator -=( int );
```

**See Also:** `WCIsvSList::forAll`, `WCIsvDList::forAll`

## ***WCIsvConstSListIter<Type>::WCIsvConstSListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WCIsvConstSListIter();`

**Semantics:** The `WCIsvConstSListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCIsvConstSListIter` public member function creates an initialized `WCIsvConstSListIter` object.

**See Also:** `WCIsvConstSListIter`, `~WCIsvConstSListIter`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
WCIsvConstSListIter( const WCIsvSList<Type> & );
```

**Semantics:** The `WCIsvConstSListIter` public member function is a constructor for the class. The value passed as a parameter is a `WCIsvSList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WCIsvConstSListIter` public member function creates an initialized `WCIsvConstSListIter` object positioned before the first element in the list.

**See Also:** `~WCIsvConstSListIter`, operator `()`, operator `++`, operator `+=`, `reset`

## ***WCIsvConstSListIter<Type>::~~WCIsvConstSListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`~WCIsvConstSListIter();`

**Semantics:** The `~WCIsvConstSListIter` public member function is the destructor for the class. The call to the `~WCIsvConstSListIter` public member function is inserted implicitly by the compiler at the point where the `WCIsvConstSListIter` object goes out of scope.

**Results:** The `WCIsvConstSListIter` object is destroyed.

**See Also:** `WCIsvConstSListIter`

**Synopsis:**

```
#include <wclistit.h>
public:
WCIsvConstDListIter();
```

**Semantics:** The `WCIsvConstDListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCIsvConstDListIter` public member function creates an initialized `WCIsvConstDListIter` object.

**See Also:** `WCIsvConstDListIter`, `~WCIsvConstDListIter`, `reset`

## ***WCIsvConstDListIter<Type>::WCIsvConstDListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WCIsvConstDListIter( const WCIsvDList<Type> & );`

**Semantics:** The `WCIsvConstDListIter` public member function is a constructor for the class. The value passed as a parameter is the `WCIsvDList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the `operator ++`, `operator ()`, or `operator +=` operators.

**Results:** The `WCIsvConstDListIter` public member function creates an initialized `WCIsvConstDListIter` object positioned before the first list element.

**See Also:** `WCIsvConstDListIter`, `~WCIsvConstDListIter`, `operator ()`, `operator ++`, `operator +=`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
~WCIsvConstDListIter();
```

**Semantics:** The `~WCIsvConstDListIter` public member function is the destructor for the class. The call to the `~WCIsvConstDListIter` public member function is inserted implicitly by the compiler at the point where the `WCIsvConstDListIter` object goes out of scope.

**Results:** The `WCIsvConstDListIter` object is destroyed.

**See Also:** `WCIsvConstDListIter`

## *WCIsvConstSListIter<Type>, WCIsvConstDListIter<Type>::container()*

---

**Synopsis:**

```
#include <wclistit.h>
public:
const WCIsvSList<Type> *WCIsvConstSListIter<Type>::container()
const;
const WCIsvDList<Type> *WCIsvConstDListIter<Type>::container()
const;
```

**Semantics:** The `container` public member function returns a pointer to the list container object. If the iterator has not been initialized with a list object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the list object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a list.

**See Also:** `WCIsvConstSListIter`, `WCIsvConstDListIter`, `reset`, `WCIterExcept::undef_iter`



## ***WClsvConstSListIter<Type>::current(), WClsvConstDListIter<Type>::current()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
Type * current();
```

**Semantics:** The `current` public member function returns a pointer to the list item at the current iterator position.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** A pointer to the current list element is returned. If the current element is undefined, `NULL(0)` is returned.

**See Also:** `operator ()`, `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_item`

## *WClsvConstSListIter<Type>,WClsvConstDListIter<Type>::operator ()()*

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`Type * operator ()();`

**Semantics:** The `operator ()` public member function is the call operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## *WClsvConstSListIter<Type>,WClsvConstDListIter<Type>::operator +=()*

---

**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator +=( int );
```

**Semantics:** The `operator +=` public member function accepts an integer value that causes the iterator to move that many elements after the current item. If the iterator was positioned before the first element in the list, the operation will set the current item to be the given element in the list.

If the current item was after the last element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to increment the iterator position more than element after the end of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The `operator +=` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator --`, `operator -=`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator --();
```

**Semantics:** The `operator --` public member function is the pre-decrement operator for the class. The list element previous to the current item is set to be the new current item. If the current item was the first element in the list, the iterator is positioned before the first element in the list. If the list is empty, the iterator will be positioned before the start of the list.

If the iterator was positioned after the last element in the list, the current item will be set to the last element.

If the iterator is not associated with a list or the iterator position previous to the decrement was before the first element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator --` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is decremented past the first element of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## ***WCIsvConstDLIter<Type>::operator -=()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator -=( int );
```

**Semantics:** The `operator -=` public member function accepts an integer value that causes the iterator to move that many elements before the current item. If the iterator was positioned after the last element in the list, the operation will set the current item to be the given number of elements from the end of the list.

If the current item was before the first element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to decrement the iterator position more than one element before the beginning of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The `operator -=` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is decremented past the first element in the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator --`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

## ***WCIsvConstSListIter<Type>::reset(), WCIsvConstDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated list.

**Results:** The iterator is positioned before the first list element.

**See Also:** `WCIsvConstSListIter`, `WCIsvConstDListIter`, `container`

## ***WCIsvConstSListIter<Type>::reset(), WCIsvConstDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void WCIsvConstSListIter<Type>::reset( const WCIsvSList<Type>
& );
void WCIsvConstDListIter<Type>::reset( const WCIsvDList<Type>
& );
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified list. The iterator is positioned before the first element in the list.

**Results:** The iterator is positioned before the first element of the specified list.

**See Also:** `WCIsvConstSListIter`, `WCIsvConstDListIter`, `container`



**Declared:** wclistit.h

The WCIsvSListIter<Type> and WCIsvDListIter<Type> classes are the templated classes used to create iterator objects for single and double linked list objects. These classes can be used only for non-constant lists. The WCIsvDConstListIter<Type> and WCIsvSConstListIter<Type> classes are provided to iterate over constant lists.

In the description of each member function, the text `Type` is used to indicate the list element type specified as the template parameter.

The WCIterExcept class is a base class of the WCIsvSListIter<Type> and WCIsvDListIter<Type> classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the WCIsvSListIter<Type> and WCIsvDListIter<Type> objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### Private Member Functions

Some functionality supported by base classes of the iterator are not appropriate in the single linked list iterator classes. Setting those functions as private members in the derived class is the standard mechanism to prevent them from being invoked. The following member functions are declared in the single linked list iterator private interface:

```
Type * operator --();
Type * operator --( int );
int insert( Type * );
```

### Public Member Functions

The following member functions are declared in the public interface:

```
WCIsvSListIter();
WCIsvSListIter( WCIsvSList<Type> & );
~WCIsvSListIter();
WCIsvDListIter();
WCIsvDListIter( WCIsvDList<Type> & );
~WCIsvDListIter();
int append( Type * );
WCIsvSList<Type> *WCIsvSListIter<Type>::container() const;
WCIsvDList<Type> *WCIsvDListIter<Type>::container() const;
Type * current() const;
void reset();
void WCIsvSListIter<Type>::reset( WCIsvSList<Type> & );
void WCIsvDListIter<Type>::reset( WCIsvDList<Type> & );
```

## *WCIsvSListIter<Type>, WCIsvDListIter<Type>*

---

In the iterators for double linked lists only:

```
int insert( Type * );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
Type * operator ();  
Type * operator ++();  
Type * operator +=( int );
```

In the iterators for double linked lists only:

```
Type * operator --();  
Type * operator -=( int );
```

**See Also:** `WCIsvSList::forall`, `WCIsvDList::forall`

**Synopsis:**

```
#include <wclistit.h>
public:
WCIsvSListIter();
```

**Semantics:** The `WCIsvSListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCIsvSListIter` public member function creates an initialized `WCIsvSListIter` object.

**See Also:** `WCIsvSListIter`, `~WCIsvSListIter`, `reset`

## ***WCIsvSListIter<Type>::WCIsvSListIter()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
WCIsvSListIter( WCIsvSList<Type> & );
```

**Semantics:** The `WCIsvSListIter` public member function is a constructor for the class. The value passed as a parameter is a `WCIsvSList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WCIsvSListIter` public member function creates an initialized `WCIsvSListIter` object positioned before the first element in the list.

**See Also:** `~WCIsvSListIter`, operator `()`, operator `++`, operator `+=`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
~WCIsvSListIter();
```

**Semantics:** The `~WCIsvSListIter` public member function is the destructor for the class. The call to the `~WCIsvSListIter` public member function is inserted implicitly by the compiler at the point where the `WCIsvSListIter` object goes out of scope.

**Results:** The `WCIsvSListIter` object is destroyed.

**See Also:** `WCIsvSListIter`

## ***WCIsvDListIter<Type>::WCIsvDListIter()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
WCIsvDListIter();
```

**Semantics:** The `WCIsvDListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCIsvDListIter` public member function creates an initialized `WCIsvDListIter` object.

**See Also:** `WCIsvDListIter`, `~WCIsvDListIter`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
WCIsvDListIter( WCIsvDList<Type> & );
```

**Semantics:** The `WCIsvDListIter` public member function is a constructor for the class. The value passed as a parameter is the `WCIsvDList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WCIsvDListIter` public member function creates an initialized `WCIsvDListIter` object positioned before the first list element.

**See Also:** `WCIsvDListIter`, `~WCIsvDListIter`, operator `()`, operator `++`, operator `+=`, `reset`

## ***WCIsvDListIter<Type>::~~WCIsvDListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`~WCIsvDListIter();`

**Semantics:** The `~WCIsvDListIter` public member function is the destructor for the class. The call to the `~WCIsvDListIter` public member function is inserted implicitly by the compiler at the point where the `WCIsvDListIter` object goes out of scope.

**Results:** The `WCIsvDListIter` object is destroyed.

**See Also:** `WCIsvDListIter`



**Synopsis:**

```
#include <wclistit.h>
public:
int append( Type * );
```

**Semantics:** The `append` public member function inserts a new element into the list container object. The new element is inserted after the current iterator item.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. The element is not appended. If the `undef_iter` exception is enabled, it is thrown.

**Results:** The new element is inserted after the current iterator item. A `TRUE` value (non-zero) is returned if the `append` is successful. A `FALSE` (zero) result is returned if the `append` fails.

**See Also:** `insert`, `WCEXCEPT::out_of_memory`, `WCITEREXCEPT::undef_iter`

## ***WCIsvSListIter<Type>,WCIsvDListIter<Type>::container()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
WCIsvSList<Type> *WCIsvSListIter<Type>::container() const;
WCIsvDList<Type> *WCIsvDListIter<Type>::container() const;
```

**Semantics:** The `container` public member function returns a pointer to the list container object. If the iterator has not been initialized with a list object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the list object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a list.

**See Also:** `WCIsvSListIter`, `WCIsvDListIter`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
Type * current();
```

**Semantics:** The `current` public member function returns a pointer to the list item at the current iterator position.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** A pointer to the current list element is returned. If the current element is undefined, `NULL(0)` is returned.

**See Also:** `operator ()`, `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_item`

## ***WClsvDListIter<Type>::insert()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int insert( Type * );
```

**Semantics:** The `insert` public member function inserts a new element into the list container object. The new element is inserted before the current iterator item. This process uses the previous link in the double linked list, so the `insert` public member function is not allowed with single linked lists.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. The element is not inserted. If the `undef_iter` exception is enabled, the exception is thrown.

**Results:** The new element is inserted before the current iterator item. A `TRUE` value (non-zero) is returned if the insert is successful. A `FALSE` (zero) result is returned if the insert fails.

**See Also:** `append`, `WCExcept::out_of_memory`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## ***WCIsvSListIter<Type>,WCIsvDListIter<Type>::operator ++()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator +=( int );
```

**Semantics:** The `operator +=` public member function accepts an integer value that causes the iterator to move that many elements after the current item. If the iterator was positioned before the first element in the list, the operation will set the current item to be the given element in the list.

If the current item was after the last element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to increment the iterator position more than element after the end of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The `operator +=` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator --`, `operator -=`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

## ***WCIsvDListIter<Type>::operator --()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator --();
```

**Semantics:** The `operator --` public member function is the pre-decrement operator for the class. The list element previous to the current item is set to be the new current item. If the current item was the first element in the list, the iterator is positioned before the first element in the list. If the list is empty, the iterator will be positioned before the start of the list.

If the iterator was positioned after the last element in the list, the current item will be set to the last element.

If the iterator is not associated with a list or the iterator position previous to the decrement was before the first element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator --` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is decremented past the first element of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator -=`, `reset`, `WCIterExcept::undef_iter`



**Synopsis:**

```
#include <wclistit.h>
public:
Type * operator -=( int );
```

**Semantics:** The `operator -=` public member function accepts an integer value that causes the iterator to move that many elements before the current item. If the iterator was positioned after the last element in the list, the operation will set the current item to be the given number of elements from the end of the list.

If the current item was before the first element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to decrement the iterator position more than one element before the beginning of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The `operator -=` public member function returns a pointer to the new current item. `NULL(0)` is returned when the iterator is decremented past the first element in the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator --`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

## ***WCIsvSListIter<Type>::reset(), WCIsvDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated list.

**Results:** The iterator is positioned before the first list element.

**See Also:** `WCIsvSListIter`, `WCIsvDListIter`, `container`

**Synopsis:**

```
#include <wclistit.h>
public:
void WCIsvSListIter<Type>::reset( WCIsvSList<Type> & );
void WCIsvDListIter<Type>::reset( WCIsvDList<Type> & );
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified list. The iterator is positioned before the first element in the list.

**Results:** The iterator is positioned before the first element of the specified list.

**See Also:** `WCIsvSListIter`, `WCIsvDListIter`, `container`

**Declared:** `wclistit.h`

The `WCPtrConstSListIter<Type>` and `WCPtrConstDListIter<Type>` classes are the templated classes used to create iterator objects for constant single and double linked list objects. These classes may be used to iterate over non-constant lists, but the `WCPtrDListIter<Type>` and `WCPtrSListIter<Type>` classes provide additional functionality for only non-constant lists.

In the description of each member function, the text `Type` is used to indicate the list element type specified as the template parameter.

The `WCIterExcept` class is a base class of the `WCPtrConstSListIter<Type>` and `WCPtrConstDListIter<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCPtrConstSListIter<Type>` and `WCPtrConstDListIter<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Private Member Functions**

Some functionality supported by base classes of the iterator are not appropriate for the constant list iterator classes. Setting those functions as private members in the derived class is the standard mechanism to prevent them from being invoked.

```
int append( Type * );
int insert( Type * );
```

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCPtrConstSListIter();
WCPtrConstSListIter( const WCPtrSList<Type> & );
~WCPtrConstSListIter();
WCPtrConstDListIter();
WCPtrConstDListIter( const WCPtrDList<Type> & );
~WCPtrConstDListIter();
const WCPtrSList<Type> *WCPtrConstSListIter<Type>::container()
const;
const WCPtrDList<Type> *WCPtrConstDListIter<Type>::container()
const;
Type * current() const;
void reset();
void WCPtrConstSListIter<Type>::reset( const WCPtrSList<Type>
& );
```

```
void WPtrConstDListIter<Type>::reset( const WPtrDList<Type>
& );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
int operator ()();
int operator ++();
int operator +=( int );
```

In the iterators for double linked lists only:

```
int operator --();
int operator -=( int );
```

**See Also:** `WPtrSList::forAll`, `WPtrDList::forAll`

## ***WPtrConstSListIter<Type>::WPtrConstSListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WPtrConstSListIter();`

**Semantics:** The `WPtrConstSListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WPtrConstSListIter` public member function creates an initialized `WPtrConstSListIter` object.

**See Also:** `WPtrConstSListIter`, `~WPtrConstSListIter`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
WCPtrConstSListIter( const WCPtrSList<Type> & );
```

**Semantics:** The `WCPtrConstSListIter` public member function is a constructor for the class. The value passed as a parameter is a `WCPtrSList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WCPtrConstSListIter` public member function creates an initialized `WCPtrConstSListIter` object positioned before the first element in the list.

**See Also:** `~WCPtrConstSListIter`, operator `()`, operator `++`, operator `+=`, `reset`

## ***WPtrConstSListIter<Type>::~~WPtrConstSListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`~WPtrConstSListIter();`

**Semantics:** The `~WPtrConstSListIter` public member function is the destructor for the class. The call to the `~WPtrConstSListIter` public member function is inserted implicitly by the compiler at the point where the `WPtrConstSListIter` object goes out of scope.

**Results:** The `WPtrConstSListIter` object is destroyed.

**See Also:** `WPtrConstSListIter`



**Synopsis:**

```
#include <wclistit.h>
public:
WPtrConstDLstIter();
```

**Semantics:** The `WPtrConstDLstIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WPtrConstDLstIter` public member function creates an initialized `WPtrConstDLstIter` object.

**See Also:** `WPtrConstDLstIter`, `~WPtrConstDLstIter`, `reset`

## ***WPtrConstDListIter<Type>::WPtrConstDListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WPtrConstDListIter( const WPtrDList<Type> & );`

**Semantics:** The `WPtrConstDListIter` public member function is a constructor for the class. The value passed as a parameter is the `WPtrDList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the `operator ++`, `operator ()`, or `operator +=` operators.

**Results:** The `WPtrConstDListIter` public member function creates an initialized `WPtrConstDListIter` object positioned before the first list element.

**See Also:** `WPtrConstDListIter`, `~WPtrConstDListIter`, `operator ()`, `operator ++`, `operator +=`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
~WCPtrConstDListIter();
```

**Semantics:** The `~WCPtrConstDListIter` public member function is the destructor for the class. The call to the `~WCPtrConstDListIter` public member function is inserted implicitly by the compiler at the point where the `WCPtrConstDListIter` object goes out of scope.

**Results:** The `WCPtrConstDListIter` object is destroyed.

**See Also:** `WCPtrConstDListIter`

## *WCPtrConstSListIter<Type>, WCPtrConstDListIter<Type>::container()*

---

**Synopsis:**

```
#include <wclistit.h>
public:
const WCPtrSList<Type> *WCPtrConstSListIter<Type>::container()
const;
const WCPtrDList<Type> *WCPtrConstDListIter<Type>::container()
const;
```

**Semantics:** The `container` public member function returns a pointer to the list container object. If the iterator has not been initialized with a list object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the list object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a list.

**See Also:** `WCPtrConstSListIter`, `WCPtrConstDListIter`, `reset`, `WCIterExcept::undef_iter`

## ***WPtrConstSListIter<Type>::current(), WPtrConstDListIter<Type>::current()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
Type * current();
```

**Semantics:** The `current` public member function returns a pointer to the list item at the current iterator position.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** A pointer to the current list element is returned. If the current element is undefined, an uninitialized pointer is returned.

**See Also:** `operator ()`, `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_item`

## *WPtrConstSListIter<Type>,WPtrConstDListIter<Type>::operator ()()*

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
int operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## *WPtrConstSListIter<Type>, WPtrConstDListIter<Type>::operator +=()*

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator +=( int );
```

**Semantics:** The `operator +=` public member function accepts an integer value that causes the iterator to move that many elements after the current item. If the iterator was positioned before the first element in the list, the operation will set the current item to be the given element in the list.

If the current item was after the last element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to increment the iterator position more than element after the end of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The `operator +=` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator --`, `operator -=`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`



**Synopsis:**

```
#include <wclistit.h>
public:
int operator --();
```

**Semantics:** The `operator --` public member function is the pre-decrement operator for the class. The list element previous to the current item is set to be the new current item. If the current item was the first element in the list, the iterator is positioned before the first element in the list. If the list is empty, the iterator will be positioned before the start of the list.

If the iterator was positioned after the last element in the list, the current item will be set to the last element.

If the iterator is not associated with a list or the iterator position previous to the decrement was before the first element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator --` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is decremented past the first element of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## ***WCPtrConstDListIter<Type>::operator -=()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator -=( int );
```

**Semantics:** The operator `-=` public member function accepts an integer value that causes the iterator to move that many elements before the current item. If the iterator was positioned after the last element in the list, the operation will set the current item to be the given number of elements from the end of the list.

If the current item was before the first element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to decrement the iterator position more than one element before the beginning of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The operator `-=` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is decremented past the first element in the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator --`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

## ***WPtrConstSListIter<Type>::reset(), WPtrConstDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated list.

**Results:** The iterator is positioned before the first list element.

**See Also:** `WPtrConstSListIter`, `WPtrConstDListIter`, `container`

## ***WPtrConstSListIter<Type>::reset(), WPtrConstDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void WPtrConstSListIter<Type>::reset( const WPtrSList<Type>
& );
void WPtrConstDListIter<Type>::reset( const WPtrDList<Type>
& );
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified list. The iterator is positioned before the first element in the list.

**Results:** The iterator is positioned before the first element of the specified list.

**See Also:** `WPtrConstSListIter`, `WPtrConstDListIter`, `container`

**Declared:** wclistit.h

The WPtrSListIter<Type> and WPtrDListIter<Type> classes are the templated classes used to create iterator objects for single and double linked list objects. These classes can be used only for non-constant lists. The WPtrDConstListIter<Type> and WPtrSConstListIter<Type> classes are provided to iterate over constant lists.

In the description of each member function, the text `Type` is used to indicate the list element type specified as the template parameter.

The WIterExcept class is a base class of the WPtrSListIter<Type> and WPtrDListIter<Type> classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the WPtrSListIter<Type> and WPtrDListIter<Type> objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### Private Member Functions

Some functionality supported by base classes of the iterator are not appropriate in the single linked list iterator classes. Setting those functions as private members in the derived class is the standard mechanism to prevent them from being invoked. The following member functions are declared in the single linked list iterator private interface:

```
int operator --();
int operator --( int );
int insert( Type * );
```

### Public Member Functions

The following member functions are declared in the public interface:

```
WPtrSListIter();
WPtrSListIter( WPtrSList<Type> & );
~WPtrSListIter();
WPtrDListIter();
WPtrDListIter( WPtrDList<Type> & );
~WPtrDListIter();
int append( Type * );
WPtrSList<Type> *WPtrSListIter<Type>::container() const;
WPtrDList<Type> *WPtrDListIter<Type>::container() const;
Type * current() const;
void reset();
void WPtrSListIter<Type>::reset( WPtrSList<Type> & );
void WPtrDListIter<Type>::reset( WPtrDList<Type> & );
```

## *WPtrSListIter<Type>, WPtrDListIter<Type>*

---

In the iterators for double linked lists only:

```
int insert( Type * );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
int operator ()();  
int operator ++();  
int operator +=( int );
```

In the iterators for double linked lists only:

```
int operator --();  
int operator -=( int );
```

**See Also:** `WPtrSList::forall`, `WPtrDList::forall`

**Synopsis:**

```
#include <wclistit.h>
public:
WCPtrSListIter();
```

**Semantics:** The `WCPtrSListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCPtrSListIter` public member function creates an initialized `WCPtrSListIter` object.

**See Also:** `WCPtrSListIter`, `~WCPtrSListIter`, `reset`

## ***WPtrSListIter<Type>::WPtrSListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WPtrSListIter( WPtrSList<Type> & );`

**Semantics:** The `WPtrSListIter` public member function is a constructor for the class. The value passed as a parameter is a `WPtrSList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WPtrSListIter` public member function creates an initialized `WPtrSListIter` object positioned before the first element in the list.

**See Also:** `~WPtrSListIter`, operator `()`, operator `++`, operator `+=`, `reset`



**Synopsis:**

```
#include <wclistit.h>
public:
~WPtrSListIter();
```

**Semantics:** The `~WPtrSListIter` public member function is the destructor for the class. The call to the `~WPtrSListIter` public member function is inserted implicitly by the compiler at the point where the `WPtrSListIter` object goes out of scope.

**Results:** The `WPtrSListIter` object is destroyed.

**See Also:** `WPtrSListIter`

## ***WPtrDLstIter<Type>::WPtrDLstIter()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
WPtrDLstIter();
```

**Semantics:** The `WPtrDLstIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WPtrDLstIter` public member function creates an initialized `WPtrDLstIter` object.

**See Also:** `WPtrDLstIter`, `~WPtrDLstIter`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
WPtrDListIter( WPtrDList<Type> & );
```

**Semantics:** The `WPtrDListIter` public member function is a constructor for the class. The value passed as a parameter is the `WPtrDList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WPtrDListIter` public member function creates an initialized `WPtrDListIter` object positioned before the first list element.

**See Also:** `WPtrDListIter`, `~WPtrDListIter`, operator `()`, operator `++`, operator `+=`, `reset`

## ***WCPtrDListIter<Type>::~~WCPtrDListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`~WCPtrDListIter();`

**Semantics:** The `~WCPtrDListIter` public member function is the destructor for the class. The call to the `~WCPtrDListIter` public member function is inserted implicitly by the compiler at the point where the `WCPtrDListIter` object goes out of scope.

**Results:** The `WCPtrDListIter` object is destroyed.

**See Also:** `WCPtrDListIter`

**Synopsis:**

```
#include <wclistit.h>
public:
int append( Type * );
```

**Semantics:** The `append` public member function inserts a new element into the list container object. The new element is inserted after the current iterator item.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. The element is not appended. If the `undef_iter` exception is enabled, it is thrown.

If the `append` fails, the `out_of_memory` exception is thrown, if enabled in the list being iterated over. The list remains unchanged.

**Results:** The new element is inserted after the current iterator item. A `TRUE` value (non-zero) is returned if the `append` is successful. A `FALSE` (zero) result is returned if the `append` fails.

**See Also:** `insert`, `WCExcept::out_of_memory`, `WCIterExcept::undef_iter`

## ***WCPtrSListIter<Type>,WCPtrDListIter<Type>::container()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WCPtrSList<Type> *WCPtrSListIter<Type>::container() const;`  
`WCPtrDList<Type> *WCPtrDListIter<Type>::container() const;`

**Semantics:** The `container` public member function returns a pointer to the list container object. If the iterator has not been initialized with a list object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the list object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a list.

**See Also:** `WCPtrSListIter`, `WCPtrDListIter`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
Type * current();
```

**Semantics:** The `current` public member function returns a pointer to the list item at the current iterator position.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** A pointer to the current list element is returned. If the current element is undefined, an uninitialized pointer is returned.

**See Also:** `operator ()`, `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_item`

## ***WCPtrDListIter<Type>::insert()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int insert( Type * );
```

**Semantics:** The `insert` public member function inserts a new element into the list container object. The new element is inserted before the current iterator item. This process uses the previous link in the double linked list, so the `insert` public member function is not allowed with single linked lists.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. The element is not inserted. If the `undef_iter` exception is enabled, the exception is thrown.

If the insert fails and the `out_of_memory` exception is enabled in the list being iterated over, the exception is thrown. The list remains unchanged.

**Results:** The new element is inserted before the current iterator item. A `TRUE` value (non-zero) is returned if the insert is successful. A `FALSE` (zero) result is returned if the insert fails.

**See Also:** `append`, `WCEXCEPT::out_of_memory`, `WCITEREXCEPT::undef_iter`



**Synopsis:**

```
#include <wclistit.h>
public:
int operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## ***WCPtrSListIter<Type>,WCPtrDListIter<Type>::operator ++()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
int operator +=( int );
```

**Semantics:** The `operator +=` public member function accepts an integer value that causes the iterator to move that many elements after the current item. If the iterator was positioned before the first element in the list, the operation will set the current item to be the given element in the list.

If the current item was after the last element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to increment the iterator position more than element after the end of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The `operator +=` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator --`, `operator -=`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

## ***WCPtrDListIter<Type>::operator --()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator --();
```

**Semantics:** The `operator --` public member function is the pre-decrement operator for the class. The list element previous to the current item is set to be the new current item. If the current item was the first element in the list, the iterator is positioned before the first element in the list. If the list is empty, the iterator will be positioned before the start of the list.

If the iterator was positioned after the last element in the list, the current item will be set to the last element.

If the iterator is not associated with a list or the iterator position previous to the decrement was before the first element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator --` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is decremented past the first element of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
int operator -=( int );
```

**Semantics:** The operator `-=` public member function accepts an integer value that causes the iterator to move that many elements before the current item. If the iterator was positioned after the last element in the list, the operation will set the current item to be the given number of elements from the end of the list.

If the current item was before the first element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to decrement the iterator position more than one element before the beginning of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The operator `-=` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is decremented past the first element in the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator --`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

## ***WPtrSListIter<Type>::reset(), WPtrDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated list.

**Results:** The iterator is positioned before the first list element.

**See Also:** `WPtrSListIter`, `WPtrDListIter`, `container`

**Synopsis:**

```
#include <wclistit.h>
public:
void WPtrSListIter<Type>::reset( WPtrSList<Type> & );
void WPtrDListIter<Type>::reset( WPtrDList<Type> & );
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified list. The iterator is positioned before the first element in the list.

**Results:** The iterator is positioned before the first element of the specified list.

**See Also:** `WPtrSListIter`, `WPtrDListIter`, `container`

**Declared:** wclistit.h

The `WCValConstSListIter<Type>` and `WCValConstDListIter<Type>` classes are the templated classes used to create iterator objects for constant single and double linked list objects. These classes may be used to iterate over non-constant lists, but the `WCValDListIter<Type>` and `WCValSListIter<Type>` classes provide additional functionality for only non-constant lists.

In the description of each member function, the text `Type` is used to indicate the list element type specified as the template parameter.

The `WCIterExcept` class is a base class of the `WCValConstSListIter<Type>` and `WCValConstDListIter<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCValConstSListIter<Type>` and `WCValConstDListIter<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Private Member Functions**

Some functionality supported by base classes of the iterator are not appropriate for the constant list iterator classes. Setting those functions as private members in the derived class is the standard mechanism to prevent them from being invoked.

```
int append( Type & );
int insert( Type & );
```

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCValConstSListIter();
WCValConstSListIter( const WCValSList<Type> & );
~WCValConstSListIter();
WCValConstDListIter();
WCValConstDListIter( const WCValDList<Type> & );
~WCValConstDListIter();
const WCValSList<Type> *WCValConstSListIter<Type>::container()
const;
const WCValDList<Type> *WCValConstDListIter<Type>::container()
const;
Type current() const;
void reset();
void WCValConstSListIter<Type>::reset( const WCValSList<Type>
& );
```



```
void WCValConstDListIter<Type>::reset( const WCValDList<Type>
& );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
int operator ()();
int operator ++();
int operator +=( int );
```

In the iterators for double linked lists only:

```
int operator --();
int operator -=( int );
```

**See Also:** `WCValSList::forAll`, `WCValDList::forAll`

## ***WCValConstSListIter<Type>::WCValConstSListIter()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
WCValConstSListIter();
```

**Semantics:** The `WCValConstSListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCValConstSListIter` public member function creates an initialized `WCValConstSListIter` object.

**See Also:** `WCValConstSListIter`, `~WCValConstSListIter`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
WCValConstSListIter( const WCValSList<Type> & );
```

**Semantics:** The `WCValConstSListIter` public member function is a constructor for the class. The value passed as a parameter is a `WCValSList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WCValConstSListIter` public member function creates an initialized `WCValConstSListIter` object positioned before the first element in the list.

**See Also:** `~WCValConstSListIter`, operator `()`, operator `++`, operator `+=`, `reset`

## ***WCValConstSListIter<Type>::~~WCValConstSListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`~WCValConstSListIter();`

**Semantics:** The `~WCValConstSListIter` public member function is the destructor for the class. The call to the `~WCValConstSListIter` public member function is inserted implicitly by the compiler at the point where the `WCValConstSListIter` object goes out of scope.

**Results:** The `WCValConstSListIter` object is destroyed.

**See Also:** `WCValConstSListIter`

**Synopsis:**

```
#include <wclistit.h>
public:
WCValConstDListIter();
```

**Semantics:** The `WCValConstDListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCValConstDListIter` public member function creates an initialized `WCValConstDListIter` object.

**See Also:** `WCValConstDListIter`, `~WCValConstDListIter`, `reset`

## ***WCValConstDListIter<Type>::WCValConstDListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WCValConstDListIter( const WCValDList<Type> & );`

**Semantics:** The `WCValConstDListIter` public member function is a constructor for the class. The value passed as a parameter is the `WCValDList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the `operator ++`, `operator ()`, or `operator +=` operators.

**Results:** The `WCValConstDListIter` public member function creates an initialized `WCValConstDListIter` object positioned before the first list element.

**See Also:** `WCValConstDListIter`, `~WCValConstDListIter`, `operator ()`, `operator ++`, `operator +=`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
~WCValConstDListIter();
```

**Semantics:** The `~WCValConstDListIter` public member function is the destructor for the class. The call to the `~WCValConstDListIter` public member function is inserted implicitly by the compiler at the point where the `WCValConstDListIter` object goes out of scope.

**Results:** The `WCValConstDListIter` object is destroyed.

**See Also:** `WCValConstDListIter`

## ***WCValConstSListIter<Type>,WCValConstDListIter<Type>::container()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
const WCValSList<Type> *WCValConstSListIter<Type>::container()
const;
const WCValDList<Type> *WCValConstDListIter<Type>::container()
const;
```

**Semantics:** The `container` public member function returns a pointer to the list container object. If the iterator has not been initialized with a list object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the list object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a list.

**See Also:** `WCValConstSListIter`, `WCValConstDListIter`, `reset`, `WCIterExcept::undef_iter`



## ***WValConstSListIter<Type>::current(), WValConstDListIter<Type>::current()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
Type current();
```

**Semantics:** The `current` public member function returns the value of the list element at the current iterator position.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** The value at the current iterator element is returned. If the current element is undefined, a default initialized object is returned.

**See Also:** `operator ()`, `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_item`

## *WCValConstSListIter<Type>,WCValConstDListIter<Type>::operator ()()*

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
int operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## ***WCValConstSListIter<Type>,WCValConstDListIter<Type>::operator +=()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator +=( int );
```

**Semantics:** The `operator +=` public member function accepts an integer value that causes the iterator to move that many elements after the current item. If the iterator was positioned before the first element in the list, the operation will set the current item to be the given element in the list.

If the current item was after the last element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to increment the iterator position more than element after the end of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The `operator +=` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator --`, `operator -=`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
int operator --();
```

**Semantics:** The `operator --` public member function is the pre-decrement operator for the class. The list element previous to the current item is set to be the new current item. If the current item was the first element in the list, the iterator is positioned before the first element in the list. If the list is empty, the iterator will be positioned before the start of the list.

If the iterator was positioned after the last element in the list, the current item will be set to the last element.

If the iterator is not associated with a list or the iterator position previous to the decrement was before the first element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator --` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is decremented past the first element of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## ***WCValConstDListIter<Type>::operator -=()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator -=( int );
```

**Semantics:** The operator `-=` public member function accepts an integer value that causes the iterator to move that many elements before the current item. If the iterator was positioned after the last element in the list, the operation will set the current item to be the given number of elements from the end of the list.

If the current item was before the first element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to decrement the iterator position more than one element before the beginning of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The operator `-=` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is decremented past the first element in the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator --`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

## ***WValConstSListIter<Type>::reset(), WValConstDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated list.

**Results:** The iterator is positioned before the first list element.

**See Also:** `WValConstSListIter`, `WValConstDListIter`, `container`

## ***WValConstSListIter<Type>::reset(), WValConstDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void WValConstSListIter<Type>::reset( const WValSList<Type>
& );
void WValConstDListIter<Type>::reset( const WValDList<Type>
& );
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified list. The iterator is positioned before the first element in the list.

**Results:** The iterator is positioned before the first element of the specified list.

**See Also:** `WValConstSListIter`, `WValConstDListIter`, `container`



**Declared:** wclistit.h

The WCValSListIter<Type> and WCValDListIter<Type> classes are the templated classes used to create iterator objects for single and double linked list objects. These classes can be used only for non-constant lists. The WCValDConstListIter<Type> and WCValSConstListIter<Type> classes are provided to iterate over constant lists.

In the description of each member function, the text `Type` is used to indicate the list element type specified as the template parameter.

The WCIterExcept class is a base class of the WCValSListIter<Type> and WCValDListIter<Type> classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the WCValSListIter<Type> and WCValDListIter<Type> objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### Private Member Functions

Some functionality supported by base classes of the iterator are not appropriate in the single linked list iterator classes. Setting those functions as private members in the derived class is the standard mechanism to prevent them from being invoked. The following member functions are declared in the single linked list iterator private interface:

```
int operator --();
int operator -=( int );
int insert( Type & );
```

### Public Member Functions

The following member functions are declared in the public interface:

```
WCValSListIter();
WCValSListIter( WCValSList<Type> & );
~WCValSListIter();
WCValDListIter();
WCValDListIter( WCValDList<Type> & );
~WCValDListIter();
int append( Type & );
WCValSList<Type> *WCValSListIter<Type>::container() const;
WCValDList<Type> *WCValDListIter<Type>::container() const;
Type current() const;
void reset();
void WCValSListIter<Type>::reset( WCValSList<Type> & );
void WCValDListIter<Type>::reset( WCValDList<Type> & );
```

## ***WCValSListIter<Type>, WCValDListIter<Type>***

---

In the iterators for double linked lists only:

```
int insert( Type & );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
int operator ()();  
int operator ++();  
int operator +=( int );
```

In the iterators for double linked lists only:

```
int operator --();  
int operator -=( int );
```

**See Also:** `WCValSList::forall`, `WCValDList::forall`

### **Sample Program Using Value List Iterators**

```
#include <wclistit.h>
#include <iostream.h>

//
// insert elem after all elements in the list less than or equal to
// elem
//

void insert_in_order( WCValDList<int> &list, int elem ) {
    if( list.entries() == 0 ) {
        // cannot insert in an empty list using a iterator
        list.insert( elem );
    } else {

        WCValDListIter<int> iter( list );
        while( ++iter ) {
            if( iter.current() > elem ) {
                // insert elem before first element in list greater
                // than elem
                iter.insert( elem );
                return;
            }
        }

        // iterated past the end of the list
        // append elem to the end of the list
        list.append( elem );
    }
}

void main() {
    WCValDList<int> list;

    insert_in_order( list, 5 );
    insert_in_order( list, 20 );
    insert_in_order( list, 1 );
    insert_in_order( list, 25 );

    cout << "List elements in ascending order:\n";

    WCValDListIter<int> iter( list );
    while( ++iter ) {
        cout << iter.current() << "\n";
    }

    cout << "List elements in descending order\n";

    // iterator is past the end of the list
    while( --iter ) {
        cout << iter.current() << "\n";
    }
}
```

## ***WCValSListIter<Type>::WCValSListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WCValSListIter();`

**Semantics:** The `WCValSListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCValSListIter` public member function creates an initialized `WCValSListIter` object.

**See Also:** `WCValSListIter`, `~WCValSListIter`, `reset`

**Synopsis:**

```
#include <wclistit.h>
public:
WCValSListIter( WCValSList<Type> & );
```

**Semantics:** The `WCValSListIter` public member function is a constructor for the class. The value passed as a parameter is a `WCValSList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WCValSListIter` public member function creates an initialized `WCValSListIter` object positioned before the first element in the list.

**See Also:** `~WCValSListIter`, operator `()`, operator `++`, operator `+=`, `reset`

## ***WCValSListIter<Type>::~~WCValSListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`~WCValSListIter();`

**Semantics:** The `~WCValSListIter` public member function is the destructor for the class. The call to the `~WCValSListIter` public member function is inserted implicitly by the compiler at the point where the `WCValSListIter` object goes out of scope.

**Results:** The `WCValSListIter` object is destroyed.

**See Also:** `WCValSListIter`

**Synopsis:**

```
#include <wclistit.h>
public:
WCVaDListIter();
```

**Semantics:** The `WCVaDListIter` public member function is the default constructor for the class and initializes the iterator with no list to operate on. The `reset` member function must be called to provide the iterator with a list to iterate over.

**Results:** The `WCVaDListIter` public member function creates an initialized `WCVaDListIter` object.

**See Also:** `WCVaDListIter`, `~WCVaDListIter`, `reset`

## ***WCValDListIter<Type>::WCValDListIter()***

---

**Synopsis:** `#include <wclistit.h>`  
`public:`  
`WCValDListIter( WCValDList<Type> & );`

**Semantics:** The `WCValDListIter` public member function is a constructor for the class. The value passed as a parameter is the `WCValDList` list object. The iterator will be initialized for that list object and positioned before the first list element. To position the iterator to a valid element within the list, increment it using any of the operator `++`, operator `()`, or operator `+=` operators.

**Results:** The `WCValDListIter` public member function creates an initialized `WCValDListIter` object positioned before the first list element.

**See Also:** `WCValDListIter`, `~WCValDListIter`, operator `()`, operator `++`, operator `+=`, `reset`



**Synopsis:**

```
#include <wclistit.h>
public:
~WCValDListIter();
```

**Semantics:** The `~WCValDListIter` public member function is the destructor for the class. The call to the `~WCValDListIter` public member function is inserted implicitly by the compiler at the point where the `WCValDListIter` object goes out of scope.

**Results:** The `WCValDListIter` object is destroyed.

**See Also:** `WCValDListIter`

## ***WCValSListIter<Type>::append(), WCValDListIter<Type>::append()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int append( Type & );
```

**Semantics:** The `append` public member function inserts a new element into the list container object. The new element is inserted after the current iterator item.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. The element is not appended. If the `undef_iter` exception is enabled, it is thrown.

If the `append` fails, the `out_of_memory` exception is thrown, if enabled in the list being iterated over. The list remains unchanged.

**Results:** The new element is inserted after the current iterator item. A `TRUE` value (non-zero) is returned if the `append` is successful. A `FALSE` (zero) result is returned if the `append` fails.

**See Also:** `insert`, `WCExcept::out_of_memory`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
WCValSList<Type> *WCValSListIter<Type>::container() const;
WCValDList<Type> *WCValDListIter<Type>::container() const;
```

**Semantics:** The `container` public member function returns a pointer to the list container object. If the iterator has not been initialized with a list object, and the `undef_iter` exception is enabled, the exception is thrown.

**Results:** A pointer to the list object associated with the iterator is returned, or `NULL(0)` if the iterator has not been initialized with a list.

**See Also:** `WCValSListIter`, `WCValDListIter`, `reset`, `WCIterExcept::undef_iter`

## *WCValSListIter<Type>::current(), WCValDListIter<Type>::current()*

---

**Synopsis:**

```
#include <wclistit.h>
public:
Type current();
```

**Semantics:** The `current` public member function returns the value of the list element at the current iterator position.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. In this case the `undef_item` exception is thrown, if enabled.

**Results:** The value at the current iterator element is returned. If the current element is undefined, a default initialized object is returned.

**See Also:** `operator ()`, `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_item`

**Synopsis:**

```
#include <wclistit.h>
public:
int insert( Type & );
```

**Semantics:** The `insert` public member function inserts a new element into the list container object. The new element is inserted before the current iterator item. This process uses the previous link in the double linked list, so the `insert` public member function is not allowed with single linked lists.

If the iterator is not associated with a list, or the iterator position is either before the first element or past the last element in the list, the current iterator position is undefined. The element is not inserted. If the `undef_iter` exception is enabled, the exception is thrown.

If the insert fails and the `out_of_memory` exception is enabled in the list being iterated over, the exception is thrown. The list remains unchanged.

**Results:** The new element is inserted before the current iterator item. A `TRUE` value (non-zero) is returned if the insert is successful. A `FALSE` (zero) result is returned if the insert fails.

**See Also:** `append`, `WCExcept::out_of_memory`, `WCIterExcept::undef_iter`

## ***WCValSListIter<Type>,WCValDListIter<Type>::operator ()()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator ()();
```

**Semantics:** The `operator ()` public member function is the call operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ()` public member function has the same semantics as the pre-increment operator, `operator ++`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ()` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `operator ++`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
int operator ++();
```

**Semantics:** The `operator ++` public member function is the pre-increment operator for the class. The list element which follows the current item is set to be the new current item. If the previous current item was the last element in the list, the iterator is positioned after the end of the list.

The `operator ++` public member function has the same semantics as the call operator, `operator ()`.

If the iterator was positioned before the first element in the list, the current item will be set to the first element in the list. If the list is empty, the iterator will be positioned after the end of the list.

If the iterator is not associated with a list or the iterator position before the increment was past the last element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator ++` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator +=`, `operator --`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## *WCValSListIter<Type>,WCValDListIter<Type>::operator +=()*

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator +=( int );
```

**Semantics:** The `operator +=` public member function accepts an integer value that causes the iterator to move that many elements after the current item. If the iterator was positioned before the first element in the list, the operation will set the current item to be the given element in the list.

If the current item was after the last element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to increment the iterator position more than element after the end of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The `operator +=` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is incremented past the end of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator --`, `operator -=`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`



**Synopsis:**

```
#include <wclistit.h>
public:
int operator --();
```

**Semantics:** The `operator --` public member function is the pre-decrement operator for the class. The list element previous to the current item is set to be the new current item. If the current item was the first element in the list, the iterator is positioned before the first element in the list. If the list is empty, the iterator will be positioned before the start of the list.

If the iterator was positioned after the last element in the list, the current item will be set to the last element.

If the iterator is not associated with a list or the iterator position previous to the decrement was before the first element the list, the `undef_iter` exception is thrown, if enabled.

**Results:** The `operator --` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is decremented past the first element of the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator -=`, `reset`, `WCIterExcept::undef_iter`

## ***WCValDListIter<Type>::operator -=()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
int operator -=( int );
```

**Semantics:** The operator `-=` public member function accepts an integer value that causes the iterator to move that many elements before the current item. If the iterator was positioned after the last element in the list, the operation will set the current item to be the given number of elements from the end of the list.

If the current item was before the first element in the list previous to the iteration, and the `undef_iter` exception is enabled, the exception will be thrown. Attempting to decrement the iterator position more than one element before the beginning of the list, or by less than one element causes the `iter_range` exception to be thrown, if enabled.

**Results:** The operator `-=` public member function returns a non-zero value if the iterator is positioned on a list item. `Zero(0)` is returned when the iterator is decremented past the first element in the list.

**See Also:** `current`, `operator ()`, `operator ++`, `operator +=`, `operator --`, `reset`, `WCIterExcept::iter_range`, `WCIterExcept::undef_iter`

**Synopsis:**

```
#include <wclistit.h>
public:
void reset();
```

**Semantics:** The `reset` public member function resets the iterator to the initial state, positioning the iterator before the first element in the associated list.

**Results:** The iterator is positioned before the first list element.

**See Also:** `WCValsListIter`, `WCValdListIter`, `container`

## ***WCValSListIter<Type>::reset(), WCValDListIter<Type>::reset()***

---

**Synopsis:**

```
#include <wclistit.h>
public:
void WCValSListIter<Type>::reset( WCValSList<Type> & );
void WCValDListIter<Type>::reset( WCValDList<Type> & );
```

**Semantics:** The `reset` public member function resets the iterator to operate on the specified list. The iterator is positioned before the first element in the list.

**Results:** The iterator is positioned before the first element of the specified list.

**See Also:** `WCValSListIter`, `WCValDListIter`, `container`

---

# 14 Queue Container

Queue containers maintain an ordered collection of data which is retrieved in the order in which the data was entered into the queue. The queue class is implemented as a templated class, allowing the use of any data type as the queue data.

A second template parameter specifies the storage class used to implement the queue. The `WCValSList`, `WCIsvSList` and `WCPtrSList` classes are appropriate storage classes.

## WCQueue<Type,FType>

---

**Declared:** `wcqueue.h`

The `WCQueue<Type , FType>` class is a templated class used to create objects which maintain data in a queue.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type of the elements stored in the queue. The text `FType` is used to indicate the template parameter defining the storage class used to maintain the queue.

For example, to create a queue of integers, the `WCQueue<int , WCValSList<int> >` class can be used. The `WCQueue<int * , WCPtrSList<int> >` class will create a queue of pointers to integers. To create an intrusive queue of objects of type `isv_link` (derived from the `WCSTLink` class), the `WCQueue< isv_link * , WCISvSList< isv_link > >` class can be used.

The `WCExcept` class is a base class of the `WCQueue<Type , FType>` class and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCQueue<Type , FType>` object. No exceptions are enabled unless they are set by the `exceptions` member function.

### Requirements of Type

`Type` must provide any constructors and/or operators required by the `FType` class.

### Public Member Functions

The following member functions are declared in the public interface:

```
WCQueue();
WCQueue( void *(*)( size_t ), void *(*)( void *, size_t ) );
~WCQueue();
void clear();
int entries() const;
Type first() const;
Type get();
int insert( const Type & );
int isEmpty() const;
Type last() const;
```

### Sample Program Using a Queue

```
#include <wcqueue.h>
#include <iostream.h>

main() {
    WCQueue<int,WCValSList<int> >    queue;
```

```
queue.insert( 7 );
queue.insert( 8 );
queue.insert( 9 );
queue.insert( 10 );

cout << "\nNumber of queue entries: " << queue.entries() << "\n";
cout << "First entry = [" << queue.first() << "]\n";
cout << "Last entry = [" << queue.last() << "]\n";
while( !queue.isEmpty() ) {
    cout << queue.get() << "\n";
};
cout.flush();
}
```

## *WCQueue<Type,FType>::WCQueue()*

---

**Synopsis:**

```
#include <wqueue.h>
public:
WCQueue();
```

**Semantics:** The public `WCQueue<Type , FType>` constructor creates an empty `WCQueue<Type , FType>` object. The `FType` storage class constructor performs the initialization.

**Results:** The public `WCQueue<Type , FType>` constructor creates an initialized `WCQueue<Type , FType>` object.

**See Also:** `~WCQueue<Type , FType>`



**Synopsis:**

```
#include <wqueue.h>
public:
WCQueue( void *(*allocator)( size_t ),
void (*deallocater)( void *, size_t ) );
```

**Semantics:** The public WCQueue<Type ,FType> constructor creates an empty WCQueue<Type ,FType> object. If FType is either the WCValSList or WCPtrSList class, then the *allocator* function is registered to perform all memory allocations of the queue elements, and the *deallocater* function to perform all freeing of the queue elements' memory. The *allocator* and *deallocater* functions are ignored if FType is the WCIsvSList class. These functions provide the ability to control how the allocation and freeing of memory is performed, allowing for more efficient memory handling than the general purpose global operator `new()` and operator `delete()` can provide. Memory management optimizations may potentially be made through the *allocator* and *deallocater* functions, but are not recommended before managing memory is understood and determined to be worth while.

The *allocator* function shall return a pointer to allocated memory of size at least the argument, or zero(0) if the allocation cannot be performed. Initialization of the memory returned is performed by the WCQueue<Type ,FType> class.

The WCQueue<Type ,FType> class calls the *deallocater* function only on memory allocated by the *allocator* function. The *deallocater* shall free the memory pointed to by the first argument which is of size the second argument. The size passed to the *deallocater* function is guaranteed to be the same size passed to the *allocator* function when the memory was allocated.

The *allocator* and *deallocater* functions may assume that for a list object instance, the *allocator* is always called with the same first argument (the size of the memory to be allocated). If FType is the WCValSList<Type> class, then the WCValSListItemSize( Type ) macro returns the size of the elements which are allocated by the *allocator* function. Similarly, the WCPtrSListItemSize( Type ) macro returns the size of WCPtrSList<Type> elements.

The FType storage class constructor performs the initialization of the queue.

**Results:** The public WCQueue<Type ,FType> constructor creates an initialized WCQueue<Type ,FType> object and registers the *allocator* and *deallocater* functions.

**See Also:** WCQueue<Type ,FType>, ~WCQueue<Type ,FType>

## ***WCQueue<Type,FType>::~~WCQueue()***

---

**Synopsis:**

```
#include <wqueue.h>
public:
virtual ~WCQueue();
```

**Semantics:** The public `~WCQueue<Type , FType>` destructor destroys the `WCQueue<Type , FType>` object. The `FType` storage class destructor performs the destruction. The call to the public `~WCQueue<Type , FType>` destructor is inserted implicitly by the compiler at the point where the `WCQueue<Type , FType>` object goes out of scope.

If the `not_empty` exception is enabled, the exception is thrown if the queue is not empty of queue elements.

**Results:** The `WCQueue<Type , FType>` object is destroyed.

**See Also:** `WCQueue<Type , FType>`, `clear`, `WCEXCEPT::not_empty`

**Synopsis:**

```
#include <wqueue.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the queue object and set it to the state of the object just after the initial construction. The queue object is not destroyed and re-created by this operator, so the object destructor is not invoked. The queue elements are not cleared by the queue class. However, the class used to maintain the queue, `FType`, may clear the items as part of the clear function for that class. If it does not clear the items, any queue items still in the list are lost unless pointed to by some pointer object in the program code.

**Results:** The `clear` public member function resets the queue object to the state of the object immediately after the initial construction.

**See Also:** `~WCQueue<Type,FType>`, `isEmpty`

## *WQueue<Type,FType>::entries()*

---

**Synopsis:**

```
#include <wqueue.h>
public:
int entries() const;
```

**Semantics:** The `entries` public member function is used to determine the number of queue elements contained in the list object.

**Results:** The number of elements in the queue is returned. Zero(0) is returned if there are no queue elements.

**See Also:** `isEmpty`

**Synopsis:**

```
#include <wqueue.h>
public:
Type first() const;
```

**Semantics:** The `first` public member function returns a queue element from the beginning of the queue object. The queue element is not removed from the queue.

If the queue is empty, one of two exceptions can be thrown. If the `empty_container` exception is enabled, then it will be thrown. Otherwise, the `index_range` exception will be thrown, if enabled.

**Results:** The first queue element is returned. If there are no elements in the queue, the return value is determined by the `find` member function of the `FType` class.

**See Also:** `get`, `isEmpty`, `last`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`, `FType::find`

## *WQueue<Type,FType>::get()*

---

**Synopsis:**

```
#include <wqueue.h>
public:
Type get();
```

**Semantics:** The `get` public member function returns the queue element which was first inserted into the queue object. The queue element is also removed from the queue.

If the queue is empty, one of two exceptions can be thrown. If the `empty_container` exception is enabled, then it will be thrown. Otherwise, the `index_range` exception will be thrown, if enabled.

**Results:** The first element in the queue is removed and returned. If there are no elements in the queue, the return value is determined by the `get` member function of the `FType` class.

**See Also:** `first`, `insert`, `isEmpty`, `WExcept::empty_container`, `WExcept::index_range`, `FType::get`

**Synopsis:**

```
#include <wqueue.h>
public:
int insert( const Type & );
```

**Semantics:** The `insert` public member function is used to insert the data into the queue. It will be the last element in the queue, and the last to be retrieved.

If the insert fails, the `out_of_memory` exception will be thrown, if enabled. The queue will remain unchanged.

**Results:** The queue element is inserted at the end of the queue. A `TRUE` value (non-zero) is returned if the insert is successful. A `FALSE` (zero) result is returned if the insert fails.

**See Also:** `get`, `WExcept::out_of_memory`

## *WQueue<Type,FType>::isEmpty()*

---

**Synopsis:**

```
#include <wqueue.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if a queue object has any queue elements contained in it.

**Results:** A TRUE value (non-zero) is returned if the queue object does not have any queue elements contained within it. A FALSE (zero) result is returned if the queue contains at least one element.

**See Also:** `entries`



**Synopsis:**

```
#include <wqueue.h>
public:
Type last() const;
```

**Semantics:** The `last` public member function returns a queue element from the end of the queue object. The queue element is not removed from the queue.

If the queue is empty, one of two exceptions can be thrown. If the `empty_container` exception is enabled, then it will be thrown. Otherwise, the `index_range` exception will be thrown, if enabled.

**Results:** The last queue element is returned. If there are no elements in the queue, the return value is determined by the `find` member function of the `FType` class.

**See Also:** `first`, `get`, `isEmpty`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`, `FType::find`



---

# ***15 Skip List Containers***

This chapter describes skip list containers.

**Declared:** wcskip.h

The `WCPtrSkipListDict<Key, Value>` class is a templated class used to store objects in a dictionary. Dictionaries store values with an associated key, which may be of any type. One example of a dictionary used in everyday life is the phone book. The phone numbers are the data values, and the customer name is the key. The equality operator of the key's type is used to locate the key-value pairs.

In the description of each member function, the text `Key` is used to indicate the template parameter defining the type of the indices pointed to by the pointers stored in the dictionary. The text `Value` is used to indicate the template parameter defining the type of the data pointed to by the pointers stored in the dictionary.

Note that pointers to the key values are stored in the dictionary. Destructors are not called on the keys pointed to. The key values pointed to in the dictionary should not be changed such that the equivalence to the old value is modified.

The iterator classes for skip lists have the same function and operator interface as the hash iterators classes. See the chapter on hash iterators for more information.

The `WCExcept` class is a base class of the `WCPtrSkipListDict<Key, Value>` class and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCPtrSkipListDict<Key, Value>` object. No exceptions are enabled unless they are set by the `exceptions` member function.

### Requirements of Key

The `WCPtrSkipListDict<Key, Value>` class requires `Key` to have:

A well defined equivalence operator with constant parameters  
(`int operator ==( const Key & ) const`).

A well defined operator less than with constant parameters  
(`int operator <( const Key & ) const`).

### Public Member Functions

The following member functions are declared in the public interface:

```
WCPtrSkipListDict( unsigned = WCSkipListDict_PROB_QUARTER,  
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS );  
WCPtrSkipListDict( unsigned = WCSkipListDict_PROB_QUARTER,  
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS, void * (*user_alloc)(
```

```
size_t size ), void (*user_dealloc)( void *old, size_t size )
);
WCPtrSkipListDict( const WCPtrSkipListDict & );
virtual ~WCPtrSkipListDict();
void clear();
void clearAndDestroy();
int contains( const Key * ) const;
unsigned entries() const;
Value * find( const Key * ) const;
Value * findKeyAndValue( const Key *, Key * & ) const;
void forAll( void (*user_fn)( Key *, Value *, void * ), void *
);
int insert( Key *, Value * );
int isEmpty() const;
Value * remove( const Key * );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
Value * & operator []( const Key * );
Value * const & operator []( const Key * ) const;
WCPtrSkipListDict & operator =( const WCPtrSkipListDict & );
int operator ==( const WCPtrSkipListDict & ) const;
```

## ***WCPtrSkipListDict<Key,Value>::WCPtrSkipListDict()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipListDict( unsigned = WCKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS );
```

**Semantics:** The public `WCPtrSkipListDict<Key,Value>` constructor creates an `WCPtrSkipListDict<Key,Value>` object with no entries. The first optional parameter, which defaults to the constant `WCKIPLIST_PROB_QUARTER`, determines the probability of having a certain number of pointers in each skip list node. The second optional parameter, which defaults to the constant `WCDEFAULT_SKIPLIST_MAX_PTRS`, determines the maximum number of pointers that are allowed in any skip list node. `WCDEFAULT_SKIPLIST_MAX_PTRS` is the maximum effective value of the second parameter. If an allocation failure occurs while creating the skip list, the `out_of_memory` exception is thrown if the `out_of_memory` exception is enabled.

**Results:** The public `WCPtrSkipListDict<Key,Value>` constructor creates an initialized `WCPtrSkipListDict<Key,Value>` object.

**See Also:** `~WCPtrSkipListDict<Key,Value>`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipListDict( unsigned = WCKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the list dictionary. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a list dictionary. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCPtrSkipListDictItemSize( Key, Value, num_of_pointers )
```

**Results:** The public `WCPtrSkipListDict<Key, Value>` constructor creates an initialized `WCPtrSkipListDict<Key, Value>` object.

**See Also:** `~WCPtrSkipListDict<Key, Value>`, `WCEXCEPT::out_of_memory`

## ***WCPtrSkipListDict<Key,Value>::WCPtrSkipListDict()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipListDict( const WCPtrSkipListDict & );
```

**Semantics:** The public `WCPtrSkipListDict<Key,Value>` constructor is the copy constructor for the `WCPtrSkipListDict<Key,Value>` class. The new skip list is created with the same probability and maximum pointers, all values or pointers stored in the list, and the exception trap states. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The public `WCPtrSkipListDict<Key,Value>` constructor creates an `WCPtrSkipListDict<Key,Value>` object which is a copy of the passed dictionary.

**See Also:** `operator =`, `WCExcept::out_of_memory`



**Synopsis:**

```
#include <wskip.h>
public:
virtual ~WCPtrSkipListDict();
```

**Semantics:** The public `~WCPtrSkipListDict<Key, Value>` destructor is the destructor for the `WCPtrSkipListDict<Key, Value>` class. If the number of dictionary elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the dictionary elements are cleared using the `clear` member function. The objects which the dictionary elements point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the public `~WCPtrSkipListDict<Key, Value>` destructor is inserted implicitly by the compiler at the point where the `WCPtrSkipListDict<Key, Value>` object goes out of scope.

**Results:** The public `~WCPtrSkipListDict<Key, Value>` destructor destroys an `WCPtrSkipListDict<Key, Value>` object.

**See Also:** `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

## ***WCPtrSkipListDict<Key,Value>::clear()***

---

**Synopsis:**

```
#include <wskip.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the dictionary so that it has no entries. Objects pointed to by the dictionary elements are not deleted. The dictionary object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the dictionary to have no elements.

**See Also:** `~WCPtrSkipListDict<Key,Value>`, `clearAndDestroy`, `operator =`

**Synopsis:**

```
#include <wskip.h>
public:
void clearAndDestroy();
```

**Semantics:** The `clearAndDestroy` public member function is used to clear the dictionary and delete the objects pointed to by the dictionary elements. The dictionary object is not destroyed and re-created by this function, so the dictionary object destructor is not invoked.

**Results:** The `clearAndDestroy` public member function clears the dictionary by deleting the objects pointed to by the dictionary elements.

**See Also:** `clear`

## ***WCPtrSkipListDict<Key,Value>::contains()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int contains( const Key * ) const;
```

**Semantics:** The `contains` public member function returns non-zero if an element with the specified key is stored in the dictionary, or zero if there is no equivalent element. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The `contains` public member function returns a non-zero value if the `Key` is found in the dictionary.

**See Also:** `find`, `findKeyAndValue`

**Synopsis:**

```
#include <wskip.h>
public:
unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to return the current number of elements stored in the dictionary.

**Results:** The `entries` public member function returns the number of elements in the dictionary.

**See Also:** `isEmpty`

## ***WCPtrSkipListDict<Key,Value>::find()***

---

**Synopsis:**

```
#include <wskip.h>
public:
Value * find( const Key * ) const;
```

**Semantics:** The `find` public member function is used to find an element with an equivalent key in the dictionary. If an equivalent element is found, a pointer to the element `Value` is returned. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element equivalent to the passed key is located in the dictionary.

**See Also:** `findKeyAndValue`

**Synopsis:**

```
#include <wskip.h>
public:
Value * findKeyAndValue( const Key *, Key * & ) const;
```

**Semantics:** The `findKeyAndValue` public member function is used to find an element in the dictionary with a key equivalent to the first parameter. If an equivalent element is found, a pointer to the element `Value` is returned. The reference to a `Key` passed as the second parameter is assigned the found element's key. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element equivalent to the passed key is located in the dictionary.

**See Also:** `findKeyAndValue`

## ***WCPtrSkipListDict<Key,Value>::forAll()***

---

**Synopsis:**

```
#include <wskip.h>
public:
void forAll(
void (*user_fn)( Key *, Value *, void * ),
void * );
```

**Semantics:** The `forAll` public member function causes the user supplied function to be invoked for every key-value pair in the dictionary. The user function has the prototype

```
void user_func( Key * key, Value * value, void * data );
```

As the elements are visited, the user function is invoked with the `Key` and `Value` components of the element passed as the first two parameters. The second parameter of the `forAll` function is passed as the third parameter to the user function. This value can be used to pass any appropriate data from the main code to the user function.

**Results:** The elements in the dictionary are all visited, with the user function being invoked for each one.

**See Also:** `find`, `findKeyAndValue`



**Synopsis:**

```
#include <wskip.h>
public:
int insert( Key *, Value * );
```

**Semantics:** The `insert` public member function inserts a key and value into the dictionary. If allocation of the node to store the key-value pair fails, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

**Results:** The `insert` public member function inserts a key and value into the dictionary. If the insert is successful, a non-zero will returned. A zero will be returned if the insert fails.

**See Also:** `operator =`, `WCExcept::out_of_memory`

## ***WCPtrSkipListDict<Key,Value>::isEmpty()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if the dictionary is empty.

**Results:** The `isEmpty` public member function returns zero if it contains at least one entry, non-zero if the dictionary is empty.

**See Also:** `entries`

**Synopsis:**

```
#include <wskip.h>
public:
Value * & operator [] ( const Key * );
```

**Semantics:** `operator []` is the dictionary index operator. A reference to the object stored in the dictionary with the given `Key` is returned. If no equivalent element is found, then a new key-value pair is created with the specified `Key` value, and initialized with the default constructor. The returned reference can then be assigned to, so that insertions can be made with the operator. If an allocation error occurs while inserting a new key-value pair, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, then a reference to address zero will be returned. This will result in a run-time error on systems which trap address zero references.

**Results:** The `operator []` public member function returns a reference to the element at the given key value. If the key does not exist, a reference to a created element is returned. The result of the operator may be assigned to.

**See Also:** `WCExcept::out_of_memory`

## ***WCPtrSkipListDict<Key,Value>::operator []()***

---

**Synopsis:**

```
#include <wskip.h>
public:
Value * const & operator [] ( const Key * ) const;
```

**Semantics:** `operator []` is the dictionary index operator. A constant reference to the object stored in the dictionary with the given `Key` is returned. If no equivalent element is found, then the `index_range` exception is thrown if it is enabled. If the exception is not enabled, then a reference to address zero will be returned. This will result in a run-time error on systems which trap address zero references.

**Results:** The `operator []` public member function returns a constant reference to the element at the given key value. The result of the operator may not be assigned to.

**See Also:** `WCEXcept::index_range`

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipListDict & operator =( const WCPtrSkipListDict & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCPtrSkipListDict<Key, Value>` class. The left hand side dictionary is first cleared using the `clear` member function, and then the right hand side dictionary is copied. The new skip list is created with the same probability and maximum pointers, all values or pointers stored in the list, and the exception trap states. If there is not enough memory to copy all of the values or pointers in the dictionary, then only some will be copied, and the `out_of_memory` exception is thrown if it is enabled. The number of entries will correctly reflect the number copied.

**Results:** The `operator =` public member function assigns the left hand side dictionary to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

## ***WCPtrSkipListDict<Key,Value>::operator ==()***

---

**Synopsis:** `#include <wskip.h>`  
`public:`  
`int operator ==( const WCPtrSkipListDict & ) const;`

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCPtrSkipListDict<Key,Value>` class. Two dictionary objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side dictionary are the same object. A FALSE (zero) value is returned otherwise.

**Synopsis:**

```
#include <wskip.h>
public:
Value * remove( const Key * );
```

**Semantics:** The `remove` public member function is used to remove the specified element from the dictionary. If an equivalent element is found, the pointer value is returned. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element is removed from the dictionary if it found.

## ***WCPtrSkipList<Type>, WCPtrSkipListSet<Type>***

---

**Declared:** `wcskip.h`

`WCPtrSkipList<Type>` and `WCPtrSkipListSet<Type>` classes are templated classes used to store objects in a skip list. A skip list is a probabilistic alternative to balanced trees, and provides a reasonable performance balance to insertion, search, and deletion. A skip list allows more than one copy of an element that is equivalent, while the skip list set allows only one copy. The equality operator of the element's type is used to locate the value.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type of the data pointed to by the pointers stored in the list.

Note that pointers to the elements are stored in the list. Destructors are not called on the elements pointed to. The data values pointed to in the list should not be changed such that the equivalence to the old value is modified.

The iterator classes for skip lists have the same function and operator interface as the hash iterators classes. See the chapter on hash iterators for more information.

The `WCEexcept` class is a base class of the `WCPtrSkipList<Type>` and `WCPtrSkipListSet<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCPtrSkipList<Type>` and `WCPtrSkipListSet<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Requirements of Type**

The `WCPtrSkipList<Type>` and `WCPtrSkipListSet<Type>` classes requires `Type` to have:

A well defined equivalence operator  
(`int operator ==( const Type & ) const`).

A well defined less than operator  
(`int operator <( const Type & ) const`).

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCPtrSkipList( unsigned = WCSKIPLIST_PROB_QUARTER, unsigned =  
WCDEFAULT_SKIPLIST_MAX_PTRS );  
WCPtrSkipList( unsigned = WCSKIPLIST_PROB_QUARTER, unsigned =  
WCDEFAULT_SKIPLIST_MAX_PTRS, void * (*user_alloc)( size_t size  
) , void (*user_dealloc)( void *old, size_t size ) );
```



```
WCPtrSkipList( const WCPtrSkipList & );
virtual ~WCPtrSkipList();
WCPtrSkipListSet( unsigned = WCSKIPLIST_PROB_QUARTER, unsigned
= WCDEFAULT_SKIPLIST_MAX_PTRS );
WCPtrSkipListSet( unsigned = WCSKIPLIST_PROB_QUARTER, unsigned
= WCDEFAULT_SKIPLIST_MAX_PTRS, void * (*user_alloc)( size_t
size ), void (*user_dealloc)( void *old, size_t size ) );
WCPtrSkipListSet( const WCPtrSkipListSet & );
virtual ~WCPtrSkipListSet();
void clear();
void clearAndDestroy();
int contains( const Type * ) const;
unsigned entries() const;
Type * find( const Type * ) const;
void forAll( void (*user_fn)( Type *, void * ) , void * );
int insert( Type * );
int isEmpty() const;
Type * remove( const Type * );
```

The following public member functions are available for the `WCPtrSkipList` class only:

```
unsigned occurrencesOf( const Type * ) const;
unsigned removeAll( const Type * );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
WCPtrSkipList & operator =( const WCPtrSkipList & );
int operator ==( const WCPtrSkipList & ) const;
WCPtrSkipListSet & operator =( const WCPtrSkipListSet & );
int operator ==( const WCPtrSkipListSet & ) const;
```

## ***WCPtrSkipListSet<Type>::WCPtrSkipListSet()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipListSet( unsigned = WCSKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS );
```

**Semantics:** The `WCPtrSkipListSet<Type>` constructor creates a `WCPtrSkipListSet` object with no entries. The first optional parameter, which defaults to the constant `WCSKIPLIST_PROB_QUARTER`, determines the probability of having a certain number of pointers in each skip list node. The second optional parameter, which defaults to the constant `WCDEFAULT_SKIPLIST_MAX_PTRS`, determines the maximum number of pointers that are allowed in any skip list node. `WCDEFAULT_SKIPLIST_MAX_PTRS` is the maximum effective value of the second parameter. If an allocation failure occurs while creating the skip list, the `out_of_memory` exception is thrown if the `out_of_memory` exception is enabled.

**Results:** The `WCPtrSkipListSet<Type>` constructor creates an initialized `WCPtrSkipListSet` object.

**See Also:** `~WCPtrSkipList<Type>`, `WCEexcept::out_of_memory`

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipListSet( unsigned = WCSKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the list. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a skip list. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCPtrSkipListSetItemSize( Type, num_of_pointers )
```

**Results:** The `WCPtrSkipListSet<Type>` constructor creates an initialized `WCPtrSkipListSet` object.

**See Also:** `~WCPtrSkipList<Type>`, `WCEXCEPT::out_of_memory`

## ***WCPtrSkipListSet<Type>::WCPtrSkipListSet()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipListSet( const WCPtrSkipListSet & );
```

**Semantics:** The `WCPtrSkipListSet<Type>` constructor is the copy constructor for the `WCPtrSkipListSet` class. The new skip list is created with the same probability and maximum pointers, all values or pointers stored in the list, and the exception trap states. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The `WCPtrSkipListSet<Type>` constructor creates a `WCPtrSkipListSet` object which is a copy of the passed list.

**See Also:** `operator =`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wskip.h>
public:
virtual ~WCPtrSkipListSet();
```

**Semantics:** The `WCPtrSkipListSet<Type>` destructor is the destructor for the `WCPtrSkipListSet` class. If the number of elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the list elements are cleared using the `clear` member function. The objects which the list elements point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the `WCPtrSkipListSet<Type>` destructor is inserted implicitly by the compiler at the point where the `WCPtrSkipListSet` object goes out of scope.

**Results:** The call to the `WCPtrSkipListSet<Type>` destructor destroys a `WCPtrSkipListSet` object.

**See Also:** `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

## ***WCPtrSkipList<Type>::WCPtrSkipList()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipList( unsigned = WCSKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS );
```

**Semantics:** The `WCPtrSkipList<Type>` constructor creates a `WCPtrSkipList` object with no entries. The first optional parameter, which defaults to the constant `WCSKIPLIST_PROB_QUARTER`, determines the probability of having a certain number of pointers in each skip list node. The second optional parameter, which defaults to the constant `WCDEFAULT_SKIPLIST_MAX_PTRS`, determines the maximum number of pointers that are allowed in any skip list node. `WCDEFAULT_SKIPLIST_MAX_PTRS` is the maximum effective value of the second parameter. If an allocation failure occurs while creating the skip list, the `out_of_memory` exception is thrown if the `out_of_memory` exception is enabled.

**Results:** The `WCPtrSkipList<Type>` constructor creates an initialized `WCPtrSkipList` object.

**See Also:** `~WCPtrSkipList<Type>`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipList( unsigned = WCKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the list. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a skip list. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCPtrSkipListItemSize( Type, num_of_pointers )
```

**Results:** The `WCPtrSkipList<Type>` constructor creates an initialized `WCPtrSkipList` object.

**See Also:** `~WCPtrSkipList<Type>`, `WCEXCEPT::out_of_memory`

## ***WCPtrSkipList<Type>::WCPtrSkipList()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipList( const WCPtrSkipList & );
```

**Semantics:** The `WCPtrSkipList<Type>` constructor is the copy constructor for the `WCPtrSkipList` class. The new skip list is created with the same probability and maximum pointers, all values or pointers stored in the list, and the exception trap states. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The `WCPtrSkipList<Type>` constructor creates a `WCPtrSkipList` object which is a copy of the passed list.

**See Also:** `operator =`, `WCEXCEPT::out_of_memory`



**Synopsis:**

```
#include <wskip.h>
public:
virtual ~WCPtrSkipList();
```

**Semantics:** The `WCPtrSkipList<Type>` destructor is the destructor for the `WCPtrSkipList` class. If the number of elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the list elements are cleared using the `clear` member function. The objects which the list elements point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the `WCPtrSkipList<Type>` destructor is inserted implicitly by the compiler at the point where the `WCPtrSkipList` object goes out of scope.

**Results:** The call to the `WCPtrSkipList<Type>` destructor destroys a `WCPtrSkipList` object.

**See Also:** `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

## ***WCPtrSkipList<Type>::clear(), WCPtrSkipListSet<Type>::clear()***

---

**Synopsis:** `#include <wskip.h>`  
`public:`  
`void clear();`

**Semantics:** The `clear` public member function is used to clear the list so that it has no entries. Objects pointed to by the list elements are not deleted. The list object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the list to have no elements.

**See Also:** `~WCPtrSkipList<Type>`, `clearAndDestroy`, `operator =`

## *WPtrSkipList<Type>,WPtrSkipListSet<Type>::clearAndDestroy()*

---

**Synopsis:**

```
#include <wskip.h>
public:
void clearAndDestroy();
```

**Semantics:** The `clearAndDestroy` public member function is used to clear the list and delete the objects pointed to by the list elements. The list object is not destroyed and re-created by this function, so the list object destructor is not invoked.

**Results:** The `clearAndDestroy` public member function clears the list by deleting the objects pointed to by the list elements, and then removing the list elements from the list.

**See Also:** `clear`

## ***WCPtrSkipList<Type>::contains(), WCPtrSkipListSet<Type>::contains()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int contains( const Type * ) const;
```

**Semantics:** The `contains` public member function returns non-zero if the element is stored in the list, or zero if there is no equivalent element. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The `contains` public member function returns a non-zero value if the element is found in the list.

**See Also:** `find`

## *WPtrSkipList<Type>::entries(), WPtrSkipListSet<Type>::entries()*

---

**Synopsis:**

```
#include <wskip.h>
public:
unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to return the current number of elements stored in the list.

**Results:** The `entries` public member function returns the number of elements in the list.

**See Also:** `isEmpty`

## ***WCPtrSkipList<Type>::find(), WCPtrSkipListSet<Type>::find()***

---

**Synopsis:**

```
#include <wskip.h>
public:
Type * find( const Type * ) const;
```

**Semantics:** The `find` public member function is used to find an element with an equivalent value in the list. If an equivalent element is found, a pointer to the element is returned. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The element equivalent to the passed value is located in the list.

**Synopsis:**

```
#include <wskip.h>
public:
void forAll(
void (*user_fn)( Type *, void * ),
void * );
```

**Semantics:** The `forAll` public member function causes the user supplied function to be invoked for every value in the list. The user function has the prototype

```
void user_func( Type * value, void * data );
```

As the elements are visited, the user function is invoked with the element passed as the first. The second parameter of the `forAll` function is passed as the second parameter to the user function. This value can be used to pass any appropriate data from the main code to the user function.

**Results:** The elements in the list are all visited, with the user function being invoked for each one.

**See Also:** `find`

## ***WCPtrSkipList<Type>::insert(), WCPtrSkipListSet<Type>::insert()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int insert( Type * );
```

**Semantics:** The `insert` public member function inserts a value into the list. If allocation of the node to store the value fails, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

With a `WCPtrSkipListSet`, there must be only one equivalent element in the set. If an element equivalent to the inserted element is already in the list set, the list set will remain unchanged, and the `not_unique` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

**Results:** The `insert` public member function inserts a value into the list. If the insert is successful, a non-zero will returned. A zero will be returned if the insert fails.

**See Also:** `operator =`, `WCEexcept::out_of_memory`, `WCEexcept::not_unique`



## ***WCPtrSkipList<Type>::isEmpty(), WCPtrSkipListSet<Type>::isEmpty()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if the list is empty.

**Results:** The `isEmpty` public member function returns zero if it contains at least one entry, non-zero if the list is empty.

**See Also:** `entries`

## ***WCPtrSkipList<Type>::occurrencesOf()***

---

**Synopsis:** `#include <wskip.h>`  
`public:`  
`unsigned occurrencesOf( const Type * ) const;`

**Semantics:** The `occurrencesOf` public member function is used to return the current number of elements stored in the list which are equivalent to the passed value. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The `occurrencesOf` public member function returns the number of elements in the list which are equivalent to the passed value.

**See Also:** `entries`, `find`, `isEmpty`

## ***WCPtrSkipList<Type>::operator =(), WCPtrSkipListSet<Type>::operator =()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCPtrSkipList & operator =( const WCPtrSkipList & );
WCPtrSkipListSet & operator =( const WCPtrSkipListSet & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCPtrSkipList<Type>` and `WCPtrSkipListSet<Type>` classes. The left hand side list is first cleared using the `clear` member function, and then the right hand side list is copied. The list function, exception trap states, and all of the list elements are copied. If there is not enough memory to copy all of the values or pointers in the list, then only some will be copied, and the `out_of_memory` exception is thrown if it is enabled. The number of entries will correctly reflect the number copied.

**Results:** The `operator =` public member function assigns the left hand side list to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

## ***WCPtrSkipList<Type>::operator ==(), WCPtrSkipListSet<Type>::operator ==()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int operator ==( const WCPtrSkipList & ) const;
int operator ==( const WCPtrSkipListSet & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCPtrSkipList<Type>` and `WCPtrSkipListSet<Type>` classes. Two list objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side list are the same object. A FALSE (zero) value is returned otherwise.

## ***WCPtrSkipList<Type>::remove(), WCPtrSkipListSet<Type>::remove()***

---

**Synopsis:**

```
#include <wskip.h>
public:
Type * remove( const Type * );
```

**Semantics:** The `remove` public member function is used to remove the specified element from the list. If an equivalent element is found, the pointer value is returned. Zero is returned if the element is not found. If the list is a `WCPtrSkipList` and there is more than one element equivalent to the specified element, then the last equivalent element added to the `WCPtrSkipList` is removed. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The element is removed from the list.

## ***WCPtrSkipList<Type>::removeAll()***

---

**Synopsis:**

```
#include <wskip.h>
public:
    unsigned removeAll( const Type * );
```

**Semantics:** The `removeAll` public member function is used to remove all elements equivalent to the specified element from the list. Zero is returned if no equivalent elements are found. Note that equivalence is based on the equivalence operator of the element type.

**Results:** All equivalent elements are removed from the list.

**Declared:** `wcskip.h`

The `WCValSkipListDict<Key, Value>` class is a templated class used to store objects in a dictionary. Dictionaries store values with an associated key, which may be of any type. One example of a dictionary used in everyday life is the phone book. The phone numbers are the data values, and the customer name is the key. The equality operator of the key's type is used to locate the key-value pairs.

In the description of each member function, the text `Key` is used to indicate the template parameter defining the type of the indices used to store data in the dictionary. The text `Value` is used to indicate the template parameter defining the type of the data stored in the dictionary.

Values are copied into the dictionary, which could be undesirable if the stored objects are complicated and copying is expensive. Value dictionaries should not be used to store objects of a base class if any derived types of different sizes would be stored in the dictionary, or if the destructor for a derived class must be called.

The iterator classes for skip lists have the same function and operator interface as the hash iterators classes. See the chapter on hash iterators for more information.

The `WCExcept` class is a base class of the `WCValSkipListDict<Key, Value>` class and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCValSkipListDict<Key, Value>` object. No exceptions are enabled unless they are set by the `exceptions` member function.

### Requirements of Key and Value

The `WCValSkipListDict<Key, Value>` class requires `Key` to have:

A default constructor (`Key::Key()`).

A well defined copy constructor (`Key::Key( const Key & )`).

A well defined assignment operator (`Key & operator =( const Key & )`).

A well defined equivalence operator with constant parameters  
(`int operator ==( const Key & ) const`).

A well defined operator less than with constant parameters  
(`int operator <( const Key & ) const`).

The `WCValSkipListDict<Key, Value>` class requires `Value` to have:

A default constructor ( Value::Value() ).

A well defined copy constructor ( Value::Value( const Value & ) ).

A well defined assignment operator ( Value & operator =( const Value & ) ).

### Public Member Functions

The following member functions are declared in the public interface:

```
WCValSkipListDict( unsigned = WCSkipListDict_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS );
WCValSkipListDict( unsigned = WCSkipListDict_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS, void * (*user_alloc)(
size_t size ), void (*user_dealloc)( void *old, size_t size )
);
WCValSkipListDict( const WCValSkipListDict & );
virtual ~WCValSkipListDict();
void clear();
int contains( const Key & ) const;
unsigned entries() const;
int find( const Key &, Value & ) const;
int findKeyAndValue( const Key &, Key &, Value & ) const;
void forAll( void (*user_fn)( Key, Value, void * ), void * );
int insert( const Key &, const Value & );
int isEmpty() const;
int remove( const Key & );
```

### Public Member Operators

The following member operators are declared in the public interface:

```
Value & operator []( const Key & );
const Value & operator []( const Key & ) const;
WCValSkipListDict & operator =( const WCValSkipListDict & );
int operator ==( const WCValSkipListDict & ) const;
```



**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipListDict( unsigned = WCSKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS );
```

**Semantics:** The public `WCValSkipListDict<Key,Value>` constructor creates an `WCValSkipListDict<Key,Value>` object with no entries. The first optional parameter, which defaults to the constant `WCSKIPLIST_PROB_QUARTER`, determines the probability of having a certain number of pointers in each skip list node. The second optional parameter, which defaults to the constant `WCDEFAULT_SKIPLIST_MAX_PTRS`, determines the maximum number of pointers that are allowed in any skip list node. `WCDEFAULT_SKIPLIST_MAX_PTRS` is the maximum effective value of the second parameter. If an allocation failure occurs while creating the skip list, the `out_of_memory` exception is thrown if the `out_of_memory` exception is enabled.

**Results:** The public `WCValSkipListDict<Key,Value>` constructor creates an initialized `WCValSkipListDict<Key,Value>` object.

**See Also:** `~WCValSkipListDict<Key,Value>`, `WCEXCEPT::out_of_memory`

## ***WCValSkipListDict<Key,Value>::WCValSkipListDict()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipListDict( unsigned = WCSKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the list dictionary. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a list dictionary. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCValSkipListDictItemSize( Key, Value, num_of_pointers )
```

**Results:** The public `WCValSkipListDict<Key,Value>` constructor creates an initialized `WCValSkipListDict<Key,Value>` object.

**See Also:** `~WCValSkipListDict<Key,Value>`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipListDict( const WCValSkipListDict & );
```

**Semantics:** The public `WCValSkipListDict<Key,Value>` constructor is the copy constructor for the `WCValSkipListDict<Key,Value>` class. The new skip list is created with the same probability and maximum pointers, all values or pointers stored in the list, and the exception trap states. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The public `WCValSkipListDict<Key,Value>` constructor creates an `WCValSkipListDict<Key,Value>` object which is a copy of the passed dictionary.

**See Also:** `operator =`, `WCEXCEPT::out_of_memory`

## ***WCValSkipListDict<Key,Value>::~~WCValSkipListDict()***

---

**Synopsis:**

```
#include <wskip.h>
public:
virtual ~WCValSkipListDict();
```

**Semantics:** The public `~WCValSkipListDict<Key,Value>` destructor is the destructor for the `WCValSkipListDict<Key,Value>` class. If the number of dictionary elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the dictionary elements are cleared using the `clear` member function. The call to the public `~WCValSkipListDict<Key,Value>` destructor is inserted implicitly by the compiler at the point where the `WCValSkipListDict<Key,Value>` object goes out of scope.

**Results:** The public `~WCValSkipListDict<Key,Value>` destructor destroys an `WCValSkipListDict<Key,Value>` object.

**See Also:** `clear`, `WCEXCEPT::not_empty`

**Synopsis:**

```
#include <wskip.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the dictionary so that it has no entries. Elements stored in the dictionary are destroyed using the destructors of `Key` and of `Value`. The dictionary object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the dictionary to have no elements.

**See Also:** `~WCValSkipListDict<Key,Value>`, `operator =`

## ***WCValSkipListDict<Key,Value>::contains()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int contains( const Key & ) const;
```

**Semantics:** The `contains` public member function returns non-zero if an element with the specified key is stored in the dictionary, or zero if there is no equivalent element. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The `contains` public member function returns a non-zero value if the `Key` is found in the dictionary.

**See Also:** `find`, `findKeyAndValue`

**Synopsis:**

```
#include <wskip.h>
public:
unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to return the current number of elements stored in the dictionary.

**Results:** The `entries` public member function returns the number of elements in the dictionary.

**See Also:** `isEmpty`

## *WCValSkipListDict<Key,Value>::find()*

---

**Synopsis:**

```
#include <wskip.h>
public:
int find( const Key &, Value & ) const;
```

**Semantics:** The `find` public member function is used to find an element with an equivalent key in the dictionary. If an equivalent element is found, a non-zero value is returned. The reference to a `Value` passed as the second argument is assigned the found element's `Value`. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element equivalent to the passed key is located in the dictionary.

**See Also:** `findKeyAndValue`



**Synopsis:**

```
#include <wskip.h>
public:
int findKeyAndValue( const Key &,
Key &, Value & ) const;
```

**Semantics:** The `findKeyAndValue` public member function is used to find an element in the dictionary with an key equivalent to the first parameter. If an equivalent element is found, a non-zero value is returned. The reference to a `Key` passed as the second parameter is assigned the found element's key. The reference to a `Value` passed as the third argument is assigned the found element's `Value`. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element equivalent to the passed key is located in the dictionary.

**See Also:** `findKeyAndValue`

## ***WCValSkipListDict<Key,Value>::forall()***

---

**Synopsis:**

```
#include <wskip.h>
public:
void forall(
void (*user_fn)( Key, Value, void * ),
void * );
```

**Semantics:** The `forall` public member function causes the user supplied function to be invoked for every key-value pair in the dictionary. The user function has the prototype

```
void user_func( Key key, Value value, void * data );
```

As the elements are visited, the user function is invoked with the `Key` and `Value` components of the element passed as the first two parameters. The second parameter of the `forall` function is passed as the third parameter to the user function. This value can be used to pass any appropriate data from the main code to the user function.

**Results:** The elements in the dictionary are all visited, with the user function being invoked for each one.

**See Also:** `find`, `findKeyAndValue`

**Synopsis:**

```
#include <wskip.h>
public:
int insert( const Key &, const Value & );
```

**Semantics:** The `insert` public member function inserts a key and value into the dictionary. If allocation of the node to store the key-value pair fails, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

**Results:** The `insert` public member function inserts a key and value into the dictionary. If the insert is successful, a non-zero will returned. A zero will be returned if the insert fails.

**See Also:** `operator =`, `WCExcept::out_of_memory`

## ***WCValSkipListDict<Key,Value>::isEmpty()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if the dictionary is empty.

**Results:** The `isEmpty` public member function returns zero if it contains at least one entry, non-zero if the dictionary is empty.

**See Also:** `entries`

**Synopsis:**

```
#include <wskip.h>
public:
Value & operator []( const Key & );
```

**Semantics:** `operator []` is the dictionary index operator. A reference to the object stored in the dictionary with the given `Key` is returned. If no equivalent element is found, then a new key-value pair is created with the specified `Key` value, and initialized with the default constructor. The returned reference can then be assigned to, so that insertions can be made with the operator. If an allocation error occurs while inserting a new key-value pair, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, then a reference to address zero will be returned. This will result in a run-time error on systems which trap address zero references.

**Results:** The `operator []` public member function returns a reference to the element at the given key value. If the key does not exist, a reference to a created element is returned. The result of the operator may be assigned to.

**See Also:** `WCEXCEPT::out_of_memory`

## ***WCValSkipListDict<Key,Value>::operator []()***

---

**Synopsis:**

```
#include <wskip.h>
public:
const Value & operator [] ( const Key & ) const;
```

**Semantics:** `operator []` is the dictionary index operator. A constant reference to the object stored in the dictionary with the given `Key` is returned. If no equivalent element is found, then the `index_range` exception is thrown if it is enabled. If the exception is not enabled, then a reference to address zero will be returned. This will result in a run-time error on systems which trap address zero references.

**Results:** The `operator []` public member function returns a constant reference to the element at the given key value. The result of the operator may not be assigned to.

**See Also:** `WCExcept::index_range`

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipListDict & operator =( const WCValSkipListDict & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCValSkipListDict<Key, Value>` class. The left hand side dictionary is first cleared using the `clear` member function, and then the right hand side dictionary is copied. The new skip list is created with the same probability and maximum pointers, all values or pointers stored in the list, and the exception trap states. If there is not enough memory to copy all of the values or pointers in the dictionary, then only some will be copied, and the `out_of_memory` exception is thrown if it is enabled. The number of entries will correctly reflect the number copied.

**Results:** The `operator =` public member function assigns the left hand side dictionary to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

## ***WCValSkipListDict<Key,Value>::operator ==( )***

---

**Synopsis:** `#include <wskip.h>`  
`public:`  
`int operator ==( const WCValSkipListDict & ) const;`

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCValSkipListDict<Key,Value>` class. Two dictionary objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side dictionary are the same object. A FALSE (zero) value is returned otherwise.



**Synopsis:**

```
#include <wskip.h>
public:
int remove( const Key & );
```

**Semantics:** The `remove` public member function is used to remove the specified element from the dictionary. If an equivalent element is found, a non-zero value is returned. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the `Key` type.

**Results:** The element is removed from the dictionary if it found.

## ***WCValSkipList<Type>, WCValSkipListSet<Type>***

---

**Declared:** `wcskip.h`

`WCValSkipList<Type>` and `WCValSkipListSet<Type>` classes are templated classes used to store objects in a skip list. A skip list is a probabilistic alternative to balanced trees, and provides a reasonable performance balance to insertion, search, and deletion. A skip list allows more than one copy of an element that is equivalent, while the skip list set allows only one copy. The equality operator of the element's type is used to locate the value.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type of the data to be stored in the list.

Values are copied into the list, which could be undesirable if the stored objects are complicated and copying is expensive. Value skip lists should not be used to store objects of a base class if any derived types of different sizes would be stored in the list, or if the destructor for a derived class must be called.

The iterator classes for skip lists have the same function and operator interface as the hash iterators classes. See the chapter on hash iterators for more information.

The `WCExcept` class is a base class of the `WCValSkipList<Type>` and `WCValSkipListSet<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCValSkipList<Type>` and `WCValSkipListSet<Type>` objects. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Requirements of Type**

The `WCValSkipList<Type>` and `WCValSkipListSet<Type>` classes requires `Type` to have:

A default constructor (`Type::Type()`).

A well defined copy constructor (`Type::Type( const Type & )`).

A well defined equivalence operator  
(`int operator ==( const Type & ) const`).

A well defined less than operator  
(`int operator <( const Type & ) const`).

### **Public Member Functions**

The following member functions are declared in the public interface:

## **504 Skip List Containers**

```
WCValSkipList( unsigned = WCSKIPLIST_PROB_QUARTER, unsigned =
WCDEFAULT_SKIPLIST_MAX_PTRS );
WCValSkipList( unsigned = WCSKIPLIST_PROB_QUARTER, unsigned =
WCDEFAULT_SKIPLIST_MAX_PTRS, void * (*user_alloc)( size_t size
), void (*user_dealloc)( void *old, size_t size ) );
WCValSkipList( const WCValSkipList & );
virtual ~WCValSkipList();
WCValSkipListSet( unsigned = WCSKIPLIST_PROB_QUARTER, unsigned =
WCDEFAULT_SKIPLIST_MAX_PTRS );
WCValSkipListSet( unsigned = WCSKIPLIST_PROB_QUARTER, unsigned =
WCDEFAULT_SKIPLIST_MAX_PTRS, void * (*user_alloc)( size_t
size ), void (*user_dealloc)( void *old, size_t size ) );
WCValSkipListSet( const WCValSkipListSet & );
virtual ~WCValSkipListSet();
void clear();
int contains( const Type & ) const;
unsigned entries() const;
int find( const Type &, Type & ) const;
void forAll( void (*user_fn)( Type, void * ), void * );
int insert( const Type & );
int isEmpty() const;
int remove( const Type & );
```

The following public member functions are available for the `WCValSkipList` class only:

```
unsigned occurrencesOf( const Type & ) const;
unsigned removeAll( const Type & );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
WCValSkipList & operator =( const WCValSkipList & );
int operator ==( const WCValSkipList & ) const;
WCValSkipListSet & operator =( const WCValSkipListSet & );
int operator ==( const WCValSkipListSet & ) const;
```

## ***WCValSkipListSet<Type>::WCValSkipListSet()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipListSet( unsigned = WCSKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS );
```

**Semantics:** The `WCValSkipListSet<Type>` constructor creates a `WCValSkipListSet` object with no entries. The first optional parameter, which defaults to the constant `WCSKIPLIST_PROB_QUARTER`, determines the probability of having a certain number of pointers in each skip list node. The second optional parameter, which defaults to the constant `WCDEFAULT_SKIPLIST_MAX_PTRS`, determines the maximum number of pointers that are allowed in any skip list node. `WCDEFAULT_SKIPLIST_MAX_PTRS` is the maximum effective value of the second parameter. If an allocation failure occurs while creating the skip list, the `out_of_memory` exception is thrown if the `out_of_memory` exception is enabled.

**Results:** The `WCValSkipListSet<Type>` constructor creates an initialized `WCValSkipListSet` object.

**See Also:** `~WCValSkipList<Type>`, `WCEexcept::out_of_memory`

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipListSet( unsigned = WCSKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the list. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a skip list. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCValSkipListSetItemSize( Type, num_of_pointers )
```

**Results:** The WCValSkipListSet<Type> constructor creates an initialized WCValSkipListSet object.

**See Also:** ~WCValSkipList<Type>, WCEXCEPT::out\_of\_memory

## ***WCValSkipListSet<Type>::WCValSkipListSet()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipListSet( const WCValSkipListSet & );
```

**Semantics:** The `WCValSkipListSet<Type>` constructor is the copy constructor for the `WCValSkipListSet` class. The new skip list is created with the same probability and maximum pointers, all values or pointers stored in the list, and the exception trap states. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The `WCValSkipListSet<Type>` constructor creates a `WCValSkipListSet` object which is a copy of the passed list.

**See Also:** `operator =`, `WCExcept::out_of_memory`

**Synopsis:**

```
#include <wskip.h>
public:
virtual ~WCValSkipListSet();
```

**Semantics:** The `WCValSkipListSet<Type>` destructor is the destructor for the `WCValSkipListSet` class. If the number of elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the list elements are cleared using the `clear` member function. The call to the `WCValSkipListSet<Type>` destructor is inserted implicitly by the compiler at the point where the `WCValSkipListSet` object goes out of scope.

**Results:** The call to the `WCValSkipListSet<Type>` destructor destroys a `WCValSkipListSet` object.

**See Also:** `clear`, `WCEXCEPT::not_empty`

## ***WCValSkipList<Type>::WCValSkipList()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipList( unsigned = WCSKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS );
```

**Semantics:** The `WCValSkipList<Type>` constructor creates a `WCValSkipList` object with no entries. The first optional parameter, which defaults to the constant `WCSKIPLIST_PROB_QUARTER`, determines the probability of having a certain number of pointers in each skip list node. The second optional parameter, which defaults to the constant `WCDEFAULT_SKIPLIST_MAX_PTRS`, determines the maximum number of pointers that are allowed in any skip list node. `WCDEFAULT_SKIPLIST_MAX_PTRS` is the maximum effective value of the second parameter. If an allocation failure occurs while creating the skip list, the `out_of_memory` exception is thrown if the `out_of_memory` exception is enabled.

**Results:** The `WCValSkipList<Type>` constructor creates an initialized `WCValSkipList` object.

**See Also:** `~WCValSkipList<Type>`, `WCEXCEPT::out_of_memory`



**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipList( unsigned = WCKIPLIST_PROB_QUARTER,
unsigned = WCDEFAULT_SKIPLIST_MAX_PTRS,
void * (*user_alloc)( size_t ),
void (*user_dealloc)( void *, size_t ) );
```

**Semantics:** Allocator and deallocator functions are specified for use when entries are inserted and removed from the list. The semantics of this constructor are the same as the constructor without the memory management functions.

The allocation function must return a zero if it cannot perform the allocation. The deallocation function is passed the size as well as the pointer to the data. Your allocation system may take advantage of the characteristic that the allocation function will always be called with the same size value for any particular instantiation of a skip list. To determine the size of the objects that the memory management functions will be required to allocate and free, the following macro may be used:

```
WCValSkipListItemSize( Type, num_of_pointers )
```

**Results:** The WCValSkipList<Type> constructor creates an initialized WCValSkipList object.

**See Also:** ~WCValSkipList<Type>, WCEXCEPT::out\_of\_memory

## ***WCValSkipList<Type>::WCValSkipList()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipList( const WCValSkipList & );
```

**Semantics:** The `WCValSkipList<Type>` constructor is the copy constructor for the `WCValSkipList` class. The new skip list is created with the same probability and maximum pointers, all values or pointers stored in the list, and the exception trap states. If there is not enough memory to copy all of the values, then only some will be copied, and the number of entries will correctly reflect the number copied. If all of the elements cannot be copied, then the `out_of_memory` exception is thrown if it is enabled.

**Results:** The `WCValSkipList<Type>` constructor creates a `WCValSkipList` object which is a copy of the passed list.

**See Also:** `operator =`, `WCExcept::out_of_memory`

**Synopsis:**

```
#include <wskip.h>
public:
virtual ~WCValSkipList();
```

**Semantics:** The `WCValSkipList<Type>` destructor is the destructor for the `WCValSkipList` class. If the number of elements is not zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the list elements are cleared using the `clear` member function. The call to the `WCValSkipList<Type>` destructor is inserted implicitly by the compiler at the point where the `WCValSkipList` object goes out of scope.

**Results:** The call to the `WCValSkipList<Type>` destructor destroys a `WCValSkipList` object.

**See Also:** `clear`, `WCEXCEPT::not_empty`

## ***WCValskipList<Type>::clear(), WCValskipListSet<Type>::clear()***

---

**Synopsis:**

```
#include <wskip.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the list so that it has no entries. Elements stored in the list are destroyed using the destructors of `Type`. The list object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the list to have no elements.

**See Also:** `~WCValskipList<Type>`, `operator =`

## *WCValskipList<Type>::contains(), WCValskipListSet<Type>::contains()*

---

**Synopsis:**

```
#include <wskip.h>
public:
int contains( const Type & ) const;
```

**Semantics:** The `contains` public member function returns non-zero if the element is stored in the list, or zero if there is no equivalent element. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The `contains` public member function returns a non-zero value if the element is found in the list.

**See Also:** `find`

## *WCVaSkipList<Type>::entries(), WCVaSkipListSet<Type>::entries()*

---

**Synopsis:**

```
#include <wskip.h>
public:
    unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to return the current number of elements stored in the list.

**Results:** The `entries` public member function returns the number of elements in the list.

**See Also:** `isEmpty`

**Synopsis:**

```
#include <wskip.h>
public:
int find( const Type &, Type & ) const;
```

**Semantics:** The `find` public member function is used to find an element with an equivalent value in the list. If an equivalent element is found, a non-zero value is returned. The reference to the element passed as the second argument is assigned the found element's value. Zero is returned if the element is not found. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The element equivalent to the passed value is located in the list.

## ***WCValSkipList<Type>::forAll(), WCValSkipListSet<Type>::forAll()***

---

**Synopsis:**

```
#include <wskip.h>
public:
void forAll(
void (*user_fn)( Type, void * ),
void * );
```

**Semantics:** The `forAll` public member function causes the user supplied function to be invoked for every value in the list. The user function has the prototype

```
void user_func( Type & value, void * data );
```

As the elements are visited, the user function is invoked with the element passed as the first. The second parameter of the `forAll` function is passed as the second parameter to the user function. This value can be used to pass any appropriate data from the main code to the user function.

**Results:** The elements in the list are all visited, with the user function being invoked for each one.

**See Also:** `find`



**Synopsis:**

```
#include <wskip.h>
public:
int insert( const Type & );
```

**Semantics:** The `insert` public member function inserts a value into the list. If allocation of the node to store the value fails, then the `out_of_memory` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

With a `WCValSkipListSet`, there must be only one equivalent element in the set. If an element equivalent to the inserted element is already in the list set, the list set will remain unchanged, and the `not_unique` exception is thrown if it is enabled. If the exception is not enabled, the insert will not be completed.

**Results:** The `insert` public member function inserts a value into the list. If the insert is successful, a non-zero will be returned. A zero will be returned if the insert fails.

**See Also:** `operator =`, `WCExcept::out_of_memory`, `WCExcept::not_unique`

## ***WCValSkipList<Type>::isEmpty(), WCValSkipListSet<Type>::isEmpty()***

---

**Synopsis:**

```
#include <wskip.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if the list is empty.

**Results:** The `isEmpty` public member function returns zero if it contains at least one entry, non-zero if the list is empty.

**See Also:** `entries`

**Synopsis:**

```
#include <wskip.h>
public:
    unsigned occurrencesOf( const Type & ) const;
```

**Semantics:** The `occurrencesOf` public member function is used to return the current number of elements stored in the list which are equivalent to the passed value. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The `occurrencesOf` public member function returns the number of elements in the list which are equivalent to the passed value.

**See Also:** `entries`, `find`, `isEmpty`

## ***WCValSkipList<Type>::operator =(), WCValSkipListSet<Type>::operator =()***

---

**Synopsis:**

```
#include <wskip.h>
public:
WCValSkipList & operator =( const WCValSkipList & );
WCValSkipListSet & operator =( const WCValSkipListSet & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCValSkipList<Type>` and `WCValSkipListSet<Type>` classes. The left hand side list is first cleared using the `clear` member function, and then the right hand side list is copied. The list function, exception trap states, and all of the list elements are copied. If there is not enough memory to copy all of the values or pointers in the list, then only some will be copied, and the `out_of_memory` exception is thrown if it is enabled. The number of entries will correctly reflect the number copied.

**Results:** The `operator =` public member function assigns the left hand side list to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

## ***WCValSkipList<Type>::operator ==( ), WCValSkipListSet<Type>::operator ==( )***

---

**Synopsis:**

```
#include <wskip.h>
public:
int operator ==( const WCValSkipList & ) const;
int operator ==( const WCValSkipListSet & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCValSkipList<Type>` and `WCValSkipListSet<Type>` classes. Two list objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side list are the same object. A FALSE (zero) value is returned otherwise.

## *WCValSkipList<Type>::remove(), WCValSkipListSet<Type>::remove()*

---

**Synopsis:**

```
#include <wskip.h>
public:
int remove( const Type & );
```

**Semantics:** The `remove` public member function is used to remove the specified element from the list. If an equivalent element is found, a non-zero value is returned. Zero is returned if the element is not found. If the list is a `WCValSkipList` and there is more than one element equivalent to the specified element, then the last equivalent element added to the `WCValSkipList` is removed. Note that equivalence is based on the equivalence operator of the element type.

**Results:** The element is removed from the list.

**Synopsis:**

```
#include <wskip.h>
public:
unsigned removeAll( const Type & );
```

**Semantics:** The `removeAll` public member function is used to remove all elements equivalent to the specified element from the list. Zero is returned if no equivalent elements are found. Note that equivalence is based on the equivalence operator of the element type.

**Results:** All equivalent elements are removed from the list.





---

# **16 Stack Container**

Stack containers maintain an ordered collection of data which is retrieved in the reverse order to which the data was entered into the stack. The stack class is implemented as a templated class, allowing the stacking of any data type.

A second template parameter specifies the storage class used to implement the stack. The `WCValSList`, `WCIsvSList` and `WCPtrSList` classes are appropriate storage classes.

## WCStack<Type,FType>

---

**Declared:** `wcstack.h`

The `WCStack<Type , FType>` class is a templated class used to create objects which maintain data in a stack.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type of the elements stored in the stack. The text `FType` is used to indicate the template parameter defining the storage class used to maintain the stack.

For example, to create a stack of integers, the `WCStack<int , WCValSList<int> >` class can be used. The `WCStack<int * , WCPtrSList<int> >` class will create a stack of pointers to integers. To create an intrusive stack of objects of type `isv_link` (derived from the `WCSTLink` class), the `WCStack< isv_link * , WCISvSList< isv_link > >` class can be used.

The `WCExcept` class is a base class of the `WCStack<Type , FType>` class and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCStack<Type , FType>` object. No exceptions are enabled unless they are set by the `exceptions` member function.

### Requirements of Type

`Type` must provide any constructors and/or operators required by the `FType` class.

### Public Member Functions

The following member functions are declared in the public interface:

```
WCStack();
WCStack( void *(*)( size_t ), void (*)( void *, size_t ) );
~WCStack();
void clear();
int entries() const;
int isEmpty() const;
Type pop();
int push( const Type & );
Type top() const;
```

### Sample Program Using a Stack

```
#include <wcstack.h>
#include <iostream.h>

void main() {
    WCStack<int,WCValSList<int> > stack;
```

```
stack.push( 7 );
stack.push( 8 );
stack.push( 9 );
stack.push( 10 );

cout << "\nNumber of stack entries: " << stack.entries() << "\n";
cout << "Top entry = [" << stack.top() << "]\n";
while( !stack.isEmpty() ) {
    cout << stack.pop() << "\n";
};
cout.flush();
}
```

## ***WCStack<Type,FType>::WCStack()***

---

**Synopsis:**

```
#include <wcstack.h>
public:
    WCStack();
```

**Semantics:** The public `WCStack<Type , FType>` constructor creates an empty `WCStack<Type , FType>` object. The `FType` storage class constructor performs the initialization.

**Results:** The public `WCStack<Type , FType>` constructor creates an initialized `WCStack<Type , FType>` object.

**See Also:** `~WCStack<Type , FType>`

**Synopsis:**

```
#include <wcstack.h>
public:
WCStack( void *(*allocator)( size_t ),
void (*deallocater)( void *, size_t ) );
```

**Semantics:** The public WCStack<Type ,FType> constructor creates an empty WCStack<Type ,FType> object. If FType is either the WCValSList or WCPtrSList class, then the *allocator* function is registered to perform all memory allocations of the stack elements, and the *deallocater* function to perform all freeing of the stack elements' memory. The *allocator* and *deallocater* functions are ignored if FType is the WCISvSList class. These functions provide the ability to control how the allocation and freeing of memory is performed, allowing for more efficient memory handling than the general purpose global operator `new()` and operator `delete()` can provide. Memory management optimizations may potentially be made through the *allocator* and *deallocater* functions, but are not recommended before managing memory is understood and determined to be worth while.

The *allocator* function shall return a pointer to allocated memory of size at least the argument, or zero(0) if the allocation cannot be performed. Initialization of the memory returned is performed by the WCStack<Type ,FType> class.

The WCStack<Type ,FType> class calls the *deallocater* function only on memory allocated by the *allocator* function. The *deallocater* shall free the memory pointed to by the first argument which is of size the second argument. The size passed to the *deallocater* function is guaranteed to be the same size passed to the *allocator* function when the memory was allocated.

The *allocator* and *deallocater* functions may assume that for a list object instance, the *allocator* is always called with the same first argument (the size of the memory to be allocated). If FType is the WCValSList<Type> class, then the `WCValSListItemSize( Type )` macro returns the size of the elements which are allocated by the *allocator* function. Similarly, the `WCPtrSListItemSize( Type )` macro returns the size of WCPtrSList<Type> elements.

The FType storage class constructor performs the initialization of the stack.

**Results:** The public WCStack<Type ,FType> constructor creates an initialized WCStack<Type ,FType> object and registers the *allocator* and *deallocater* functions.

**See Also:** WCStack<Type ,FType>, ~WCStack<Type ,FType>

## ***WCStack<Type,FType>::~~WCStack()***

---

**Synopsis:** `#include <wcstack.h>`  
`public:`  
`virtual ~WCStack();`

**Semantics:** The public `~WCStack<Type , FType>` destructor destroys the `WCStack<Type , FType>` object. The `FType` storage class destructor performs the destruction. The call to the public `~WCStack<Type , FType>` destructor is inserted implicitly by the compiler at the point where the `WCStack<Type , FType>` object goes out of scope.

If the `not_empty` exception is enabled, the exception is thrown if the stack is not empty of stack elements.

**Results:** The `WCStack<Type , FType>` object is destroyed.

**See Also:** `WCStack<Type , FType>`, `clear`, `WCEXCEPT::not_empty`

**Synopsis:**

```
#include <wcstack.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the stack object and set it to the state of the object just after the initial construction. The stack object is not destroyed and re-created by this operator, so the object destructor is not invoked. The stack elements are not cleared by the stack class. However, the class used to maintain the stack, `FType`, may clear the items as part of the `clear` member function for that class. If it does not clear the items, any stack items still in the list are lost unless pointed to by some pointer object in the program code.

**Results:** The `clear` public member function resets the stack object to the state of the object immediately after the initial construction.

**See Also:** `~WCStack<Type,FType>`, `isEmpty`

## ***WCStack<Type,FType>::entries()***

---

**Synopsis:**

```
#include <wcstack.h>
public:
int entries() const;
```

**Semantics:** The `entries` public member function is used to determine the number of stack elements contained in the list object.

**Results:** The number of elements on the stack is returned. `Zero(0)` is returned if there are no stack elements.

**See Also:** `isEmpty`



**Synopsis:**

```
#include <wcstack.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if a stack object has any stack elements contained in it.

**Results:** A TRUE value (non-zero) is returned if the stack object does not have any stack elements contained within it. A FALSE (zero) result is returned if the stack contains at least one element.

**See Also:** `entries`

## *WCStack<Type,FType>::pop()*

---

**Synopsis:**

```
#include <wstack.h>
public:
Type pop();
```

**Semantics:** The `pop` public member function returns the top stack element from the stack object. The top stack element is the last element pushed onto the stack. The stack element is also removed from the stack.

If the stack is empty, one of two exceptions can be thrown. If the `empty_container` exception is enabled, then it will be thrown. Otherwise, the `index_range` exception will be thrown, if enabled.

**Results:** The top stack element is removed and returned. The return value is determined by the `get` member function of the `FType` class if there are no elements on the stack.

**See Also:** `isEmpty`, `push`, `top`, `WExcept::empty_container`, `WExcept::index_range`, `FType::get`

**Synopsis:**

```
#include <wcstack.h>
public:
int push( const Type & );
```

**Semantics:** The `push` public member function is used to push the data onto the top of the stack. It will be the first element on the stack to be popped.

If the push fails, the `out_of_memory` exception will be thrown, if enabled, and the stack will remain unchanged.

**Results:** The stack element is pushed onto the top of the stack. A `TRUE` value (non-zero) is returned if the push is successful. A `FALSE` (zero) result is returned if the push fails.

**See Also:** `pop`, `WCExcept::out_of_memory`

## *WCStack<Type,FType>::top()*

---

**Synopsis:**

```
#include <wcstack.h>
public:
Type top() const;
```

**Semantics:** The `top` public member function returns the top stack element from the stack object. The top stack element is the last element pushed onto the stack. The stack element is not removed from the stack.

If the stack is empty, one of two exceptions can be thrown. If the `empty_container` exception is enabled, then it will be thrown. Otherwise, the `index_range` exception will be thrown, if enabled.

**Results:** The top stack element is returned. The return value is determined by the `find` member function of the `FType` class if there are no elements on the stack.

**See Also:** `isEmpty`, `pop`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`, `FType::find`

---

# ***17 Vector Containers***

This chapter describes vector containers.

**Declared:** `wcvector.h`

The `WCPtrSortedVector<Type>` and `WCPtrOrderedVector<Type>` classes are templated classes used to store objects in a vector. Ordered and Sorted vectors are powerful arrays which can be resized and provide an abstract interface to insert, find and remove elements. An ordered vector maintains the order in which elements are added, and allows more than one copy of an element that is equivalent. The sorted vector allow only one copy of an equivalent element, and inserts them in a sorted order. The sorted vector is less efficient when inserting elements, but can provide a faster retrieval time.

Elements cannot be inserted into these vectors by assigning to a vector index. Vectors automatically grow when necessary to insert an element if the `resize_required` exception is not enabled.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type pointed to by the pointers stored in the vector.

Note that lookups are performed on the types pointed to, not just by comparing pointers. Two pointer elements are equivalent if the values they point to are equivalent. The values pointed to do not need to be the same object.

The `WCPtrOrderedVector` class stores elements in the order which they are inserted using the `insert`, `append`, `prepend` and `insertAt` member functions. Linear searches are performed to locate entries, and the less than operator is not required.

The `WCPtrSortedVector` class stores elements in ascending order. This requires that `Type` provides a less than operator. Insertions are more expensive than inserting or appending into an ordered vector, since entries must be moved to make room for the new element. A binary search is used to locate elements in a sorted vector, making searches quicker than in the ordered vector.

Care must be taken when using the `WCPtrSortedVector` class not to change the ordering of the vector elements. An object pointed to by a vector element must not be changed so that it is not equivalent to the value when the pointer was inserted into the vector. The index operator and the member functions `find`, `first`, and `last` all return pointers the elements pointed to by the vector elements. Lookups assume elements are in sorted order, so you should not use the returned pointers to change the ordering of the value pointed to.

The `WCPtrVector` class is also available. It provides a resizable and boundary safe vector similar to standard arrays.

The `WCExcept` class is a base class of the `WCPtrSortedVector<Type>` and `WCPtrOrderedVector<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the

WCPtrSortedVector<Type> and WCPtrOrderedVector<Type> objects. No exceptions are enabled unless they are set by the exceptions member function.

### **Requirements of Type**

Both the WCPtrSortedVector<Type> and WCPtrOrderedVector<Type> classes require Type to have:

A well defined equivalence operator with constant parameters  
(int operator ==( const Type & ) const).

Additionally the WCPtrSortedVector class requires Type to have:

A well defined less than operator with constant parameters  
(int operator <( const Type & ) const).

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCPtrOrderedVector( size_t = WCDEFAULT_VECTOR_LENGTH, unsigned
= WCDEFAULT_VECTOR_RESIZE_GROW );
WCPtrOrderedVector( const WCPtrOrderedVector & );
virtual ~WCPtrOrderedVector();
WCPtrSortedVector( size_t = WCDEFAULT_VECTOR_LENGTH, unsigned
= WCDEFAULT_VECTOR_RESIZE_GROW );
WCPtrSortedVector( const WCPtrSortedVector & );
virtual ~WCPtrSortedVector();
void clear();
void clearAndDestroy();
int contains( const Type * ) const;
unsigned entries() const;
Type * find( const Type * ) const;
Type * first() const;
int index( const Type * ) const;
int insert( Type * );
int isEmpty() const;
Type * last() const;
int occurrencesOf( const Type * ) const;
Type * remove( const Type * );
unsigned removeAll( const Type * );
Type * removeAt( int );
Type * removeFirst();
Type * removeLast();
int resize( size_t );
```

The following public member functions are available for the `WPtrOrderedVector` class only:

```
int append( Type * );
int insertAt( int, Type * );
int prepend( Type * );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
Type * & operator []( int );
Type * const & operator []( int ) const;
WPtrOrderedVector & WPtrOrderedVector::operator =( const
WPtrOrderedVector & );
WPtrSortedVector & WPtrSortedVector::operator =( const
WPtrSortedVector & );
int WPtrOrderedVector::operator ==( const WPtrOrderedVector
& ) const;
int WPtrSortedVector::operator ==( const WPtrSortedVector &
) const;
```



**Synopsis:**

```
#include <wcvector.h>
public:
WCPtrOrderedVector( size_t = WCDEFAULT_VECTOR_LENGTH,
unsigned = WCDEFAULT_VECTOR_RESIZE_GROW );
```

**Semantics:** The `WCPtrOrderedVector<Type>` constructor creates an empty `WCPtrOrderedVector` object able to store the number of elements specified in the first optional parameter, which defaults to the constant `WCDEFAULT_VECTOR_LENGTH` (currently defined as 10). If the `resize_required` exception is not enabled, then the second optional parameter is used to specify the value to increase the vector size when an element is inserted into a full vector. If `zero(0)` is specified as the second parameter, any attempt to insert into a full vector fails. This parameter defaults to the constant `WCDEFAULT_VECTOR_RESIZE_GROW` (currently defined as 5).

If the vector object cannot be fully initialized, the vector is created with length zero.

**Results:** The `WCPtrOrderedVector<Type>` constructor creates an empty initialized `WCPtrOrderedVector` object.

**See Also:** `WCExcept::resize_required`

## ***WPtrOrderedVector<Type>::WPtrOrderedVector()***

---

**Synopsis:**

```
#include <wvector.h>
public:
WPtrOrderedVector( const WPtrOrderedVector & );
```

**Semantics:** The `WPtrOrderedVector<Type>` constructor is the copy constructor for the `WPtrOrderedVector` class. The new vector is created with the same length and `resize` value as the passed vector. All of the vector elements and exception trap states are copied.

If the new vector cannot be fully created, it will have length zero. The `out_of_memory` exception is thrown if enabled in the vector being copied.

**Results:** The `WPtrOrderedVector<Type>` creates a `WPtrOrderedVector` object which is a copy of the passed vector.

**See Also:** `operator =`, `WExcept::out_of_memory`

**Synopsis:**

```
#include <wvector.h>
public:
virtual ~WPtrOrderedVector();
```

**Semantics:** The `WPtrOrderedVector<Type>` destructor is the destructor for the `WPtrOrderedVector` class. If the vector is not length zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the vector entries are cleared using the `clear` member function. The objects which the vector entries point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the `WPtrOrderedVector<Type>` destructor is inserted implicitly by the compiler at the point where the `WPtrOrderedVector` object goes out of scope.

**Results:** The `WPtrOrderedVector<Type>` destructor destroys an `WPtrOrderedVector` object.

**See Also:** `clear`, `clearAndDestroy`, `WExcept::not_empty`

## ***WPtrSortedVector<Type>::WPtrSortedVector()***

---

**Synopsis:**

```
#include <wvector.h>
public:
WPtrSortedVector( size_t = WCDEFAULT_VECTOR_LENGTH,
unsigned = WCDEFAULT_VECTOR_RESIZE_GROW );
```

**Semantics:** The `WPtrSortedVector<Type>` constructor creates an empty `WPtrSortedVector` object able to store the number of elements specified in the first optional parameter, which defaults to the constant `WCDEFAULT_VECTOR_LENGTH` (currently defined as 10). If the `resize_required` exception is not enabled, then the second optional parameter is used to specify the value to increase the vector size when an element is inserted into a full vector. If zero(0) is specified as the second parameter, any attempt to insert into a full vector fails. This parameter defaults to the constant `WCDEFAULT_VECTOR_RESIZE_GROW` (currently defined as 5).

If the vector object cannot be fully initialized, the vector is created with length zero.

**Results:** The `WPtrSortedVector<Type>` constructor creates an empty initialized `WPtrSortedVector` object.

**See Also:** `WExcept::resize_required`

**Synopsis:**

```
#include <wvector.h>
public:
WPtrSortedVector( const WPtrSortedVector & );
```

**Semantics:** The `WPtrSortedVector<Type>` constructor is the copy constructor for the `WPtrSortedVector` class. The new vector is created with the same length and resize value as the passed vector. All of the vector elements and exception trap states are copied.

If the new vector cannot be fully created, it will have length zero. The `out_of_memory` exception is thrown if enabled in the vector being copied.

**Results:** The `WPtrSortedVector<Type>` constructor creates a `WPtrSortedVector` object which is a copy of the passed vector.

**See Also:** `operator =`, `WExcept::out_of_memory`

## ***WCPtrSortedVector<Type>::~~WCPtrSortedVector()***

---

**Synopsis:**

```
#include <wvector.h>
public:
virtual ~WCPtrSortedVector();
```

**Semantics:** The `WCPtrSortedVector<Type>` destructor is the destructor for the `WCPtrSortedVector` class. If the vector is not length zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the vector entries are cleared using the `clear` member function. The objects which the vector entries point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the `WCPtrSortedVector<Type>` destructor is inserted implicitly by the compiler at the point where the `WCPtrSortedVector` object goes out of scope.

**Results:** The `WCPtrSortedVector<Type>` destructor destroys an `WCPtrSortedVector` object.

**See Also:** `clear`, `clearAndDestroy`, `WCEXCEPT::not_empty`

**Synopsis:**

```
#include <wvector.h>
public:
int append( Type * );
```

**Semantics:** The `append` public member function appends the passed element to be the last element in the vector. This member function has the same semantics as the `WCPtrOrderedVector::insert` member function.

This function is not provided by the `WCPtrSortedVector` class, since all elements must be inserted in sorted order by the `insert` member function.

Several different results can occur if the vector is not large enough for the new element. If the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the `append` fails if the amount the vector is to be grown (the second parameter to the constructor) is `zero(0)`. Otherwise, the vector is automatically grown by the number of elements specified to the constructor, using the `resize` member function. If `resize` fails, the element is not appended to the vector and the `out_of_memory` exception is thrown, if enabled.

**Results:** The `append` public member function appends an element to the `WCPtrOrderedVector` object. A `TRUE` (non-zero) value is returned if the `append` is successful. If the `append` fails, a `FALSE` (zero) value is returned.

**See Also:** `insert`, `insertAt`, `prepend`, `WCEXCEPT::out_of_memory`, `WCEXCEPT::resize_required`

## ***WPtrSortedVector<Type>::clear(), WPtrOrderedVector<Type>::clear()***

---

**Synopsis:**

```
#include <wvector.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the vector so that it contains no entries, and is zero size. Objects pointed to by the vector elements are not deleted. The vector object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the vector to have zero length and no entries.

**See Also:** `~WPtrOrderedVector`, `clearAndDestroy`, `operator =`



## ***WPtrSortedVector<Type>,WPtrOrderedVector<Type>::clearAndDestroy()***

---

**Synopsis:**

```
#include <wvector.h>
public:
void clearAndDestroy();
```

**Semantics:** The `clearAndDestroy` public member function is used to clear the vector to have zero length and delete the objects pointed to by the vector elements. The vector object is not destroyed and re-created by this function, so the vector object destructor is not invoked.

**Results:** The `clearAndDestroy` public member function clears the vector by deleting the objects pointed to by the vector elements and makes the vector zero length.

**See Also:** `clear`

## ***WPtrSortedVector<Type>,WPtrOrderedVector<Type>::contains()***

---

**Synopsis:**

```
#include <wvector.h>
public:
int contains( const Type * ) const;
```

**Semantics:** The `contains` public member function is used to determine if a value is contained by a vector. Note that comparisons are done on the objects pointed to, not the pointers themselves. A linear search is used by the `WPtrOrderedVector` class to find the value. The `WPtrSortedVector` class uses a binary search.

**Results:** The `contains` public member function returns a `TRUE` (non-zero) value if the element is found in the vector. A `FALSE` (zero) value is returned if the vector does not contain the element.

**See Also:** `index`, `find`

## ***WPtrSortedVector<Type>::entries(), WPtrOrderedVector<Type>::entries()***

---

**Synopsis:**

```
#include <wvector.h>
public:
    unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to find the number of elements which are stored in the vector.

**Results:** The `entries` public member function returns the number of elements in the vector.

**See Also:** `isEmpty`

## ***WCPtrSortedVector<Type>::find(), WCPtrOrderedVector<Type>::find()***

---

**Synopsis:**

```
#include <wvector.h>
public:
Type * find( const Type * ) const;
```

**Semantics:** The `find` public member function is used to find an element equivalent to the element passed. Note that comparisons are done on the objects pointed to, not the pointers themselves. The `WCPtrOrderedVector` class uses a linear search to find the element, and the `WCPtrSortedVector` class uses a binary search.

**Results:** A pointer to the first equivalent element is returned. `NULL(0)` is returned if the element is not in the vector.

**See Also:** `contains`, `first`, `index`, `last`, `occurrencesOf`, `remove`

## ***WCPtrSortedVector<Type>::first(), WCPtrOrderedVector<Type>::first()***

---

**Synopsis:**

```
#include <wvector.h>
public:
Type * first() const;
```

**Semantics:** The `first` public member function returns the first element in the vector. The element is not removed from the vector.

If the vector is empty, one of two exceptions can be thrown. The `empty_container` exception is thrown if it is enabled. Otherwise, if the `index_range` exception is enabled, it is thrown. If neither exception is enabled, a first element of the vector is added with a NULL value.

**Results:** The `first` public member function returns the value of the first element in the vector.

**See Also:** `last`, `removeFirst`, `WCEexcept::index_range`,  
`WCEexcept::resize_required`

## ***WCPtrSortedVector<Type>::index(), WCPtrOrderedVector<Type>::index()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int index( const Type * ) const;
```

**Semantics:** The `index` public member function is used find the index of the first element equivalent to the passed element. Note that comparisons are done on the objects pointed to, not the pointers themselves. A linear search is used by the `WCPtrOrderedVector` class to find the element. The `WCPtrSortedVector` class uses a binary search.

**Results:** The `index` public member function returns the index of the first element equivalent to the parameter. If the passed value is not contained in the vector, negative one (-1) is returned.

**See Also:** `contains`, `find`, `insertAt`, `operator []`, `removeAt`

## ***WCPtrSortedVector<Type>::insert(), WCPtrOrderedVector<Type>::insert()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int insert( Type * );
```

**Semantics:** The `insert` public member function inserts the value into the vector.

The `WCPtrOrderedVector::insert` function inserts the value as the last element of the vector, and has the same semantics as the `WCPtrOrderedVector::append` member function.

A binary search is performed to determine where the value should be inserted for the `WCPtrSortedVector::insert` function. Note that comparisons are done on the objects pointed to, not the pointers themselves. Any elements greater than the inserted value are copied up one index so that the new element is after all elements with value less than or equal to it.

Several different results can occur if the vector is not large enough for the new element. If the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the insert fails if the amount the vector is to be grown (the second parameter to the constructor) is zero(0). Otherwise, the vector is automatically grown by the number of elements specified to the constructor, using the `resize` member function. If `resize` fails, the element is not inserted to the vector and the `out_of_memory` exception is thrown, if enabled.

**Results:** The `insert` public member function inserts an element in to the vector. A TRUE (non-zero) value is returned if the insert is successful. If the insert fails, a FALSE (zero) value is returned.

**See Also:** `append`, `insertAt`, `prepend`, `WCEexcept::out_of_memory`, `WCEexcept::resize_required`

## ***WCPtrOrderedVector<Type>::insertAt()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int insertAt( int, Type * );
```

**Semantics:** The `insertAt` public member function inserts the second argument into the vector before the element at index given by the first argument. If the passed index is equal to the number of entries in the vector, the new value is appended to the vector as the last element. All vector elements with indexes greater than or equal to the first parameter are copied up one index.

This function is not provided by the `WCPtrSortedVector` class, since all elements must be inserted in sorted order by the `insert` member function.

If the passed index is negative or greater than the number of entries in the vector and the `index_range` exception is enabled, the exception is thrown. If the exception is not enabled, the new element is inserted as the first element when the index is negative, or as the last element when the index is too large.

Several different results can occur if the vector is not large enough for the new element. If the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the insert fails if the amount the vector is to be grown (the second parameter to the constructor) is zero(0). Otherwise, the vector is automatically grown by the number of elements specified to the constructor, using the `resize` member function. If `resize` fails, the element is not inserted into the vector and the `out_of_memory` exception is thrown, if enabled.

**Results:** The `insertAt` public member function inserts an element into the `WCPtrOrderedVector` object before the element at the given index. A TRUE (non-zero) value is returned if the insert is successful. If the insert fails, a FALSE (zero) value is returned.

**See Also:** `append`, `insert`, `prepend`, `operator []`, `removeAt`, `WCEXCEPT::index_range`, `WCEXCEPT::out_of_memory`, `WCEXCEPT::resize_required`



## *WPtrSortedVector<Type>, WPtrOrderedVector<Type>::isEmpty()*

---

**Synopsis:**

```
#include <wvector.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if a vector object has any entries contained in it.

**Results:** A TRUE value (non-zero) is returned if the vector object does not have any vector elements contained within it. A FALSE (zero) result is returned if the vector contains at least one element.

**See Also:** `entries`

## ***WCPtrSortedVector<Type>::last(), WCPtrOrderedVector<Type>::last()***

---

**Synopsis:**

```
#include <wvector.h>
public:
Type * last() const;
```

**Semantics:** The `last` public member function returns the last element in the vector. The element is not removed from the vector.

If the vector is empty, one of two exceptions can be thrown. The `empty_container` exception is thrown if it is enabled. Otherwise, if the `index_range` exception is enabled, it is thrown. If neither exception is enabled, a first element of the vector is added with a NULL value.

**Results:** The `last` public member function returns the value of the last element in the vector.

**See Also:** `first`, `removeLast`, `WCEexcept::index_range`,  
`WCEexcept::resize_required`

## ***WPtrSortedVector<Type>,WPtrOrderedVector<Type>::occurrencesOf()***

---

**Synopsis:**

```
#include <wvector.h>
public:
int occurrencesOf( const Type * ) const;
```

**Semantics:** The `occurrencesOf` public member function returns the number of elements contained in the vector that are equivalent to the passed value. Note that comparisons are done on the objects pointed to, not the pointers themselves. A linear search is used by the `WPtrOrderedVector` class to find the value. The `WPtrSortedVector` class uses a binary search.

**Results:** The `occurrencesOf` public member function returns the number of elements equivalent to the passed value.

**See Also:** `contains`, `find`, `index`, `operator []`, `removeAll`

## *WCPtrSortedVector<Type>,WCPtrOrderedVector<Type>::operator []()*

---

**Synopsis:**

```
#include <wvector.h>
public:
Type * & operator [] ( int );
Type * const & operator [] ( int ) const;
```

**Semantics:** `operator []` is the vector index operator. A reference to the object stored in the vector at the given index is returned. If a constant vector is indexed, a reference to a constant element is returned.

The `append`, `insert`, `insertAt` and `prepend` member functions are used to insert a new element into a vector, and the `remove`, `removeAll`, `removeAt`, `removeFirst` and `removeLast` member functions remove elements. The index operator cannot be used to change the number of entries in the vector. Searches may be performed using the `find` and `index` member functions.

If the vector is empty, one of two exceptions can be thrown. The `empty_container` exception is thrown if it is enabled. Otherwise, if the `index_range` exception is enabled, it is thrown. If neither exception is enabled, a first element of the vector is added with a NULL value. This element is added so that a reference to a valid vector element can be returned.

If the index value is negative and the `index_range` exception is enabled, the exception is thrown. An attempt to index an element with `index` greater than or equal to the number of entries in the vector will also cause the `index_range` exception to be thrown if enabled. If the exception is not enabled, attempting to index a negative element will index the first element in the vector, and attempting to index an element after the last entry will index the last element.

Care must be taken when using the `WCPtrSortedVector` class not to change the ordering of the vector elements. The result returned by the index operator must not be assigned to or modified in such a way that it is no longer equivalent (by `Type`'s equivalence operator) to the value inserted into the vector. Failure to comply may cause lookups to work incorrectly, since the binary search algorithm assumes elements are in sorted order.

**Results:** The `operator []` public member function returns a reference to the element at the given index. If the index is invalid, a reference to the closest valid element is returned. The result of the non-constant index operator may be assigned to.

**See Also:** `append`, `find`, `first`, `index`, `insert`, `insertAt`, `isEmpty`, `last`, `prepend`, `remove`, `removeAt`, `removeAll`, `removeFirst`, `removeLast`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`

**Synopsis:**

```
#include <wvector.h>
public:
WPtrOrderedVector & WPtrOrderedVector::operator =( const
WPtrOrderedVector & );
WPtrSortedVector & WPtrSortedVector::operator =( const
WPtrSortedVector & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the class. The left hand side vector is first cleared using the `clear` member function, and then the right hand side vector is copied. The left hand side vector is made to have the same length and growth amount as the right hand side (the growth amount is the second argument passed to the right hand side vector constructor). All of the vector elements and exception trap states are copied.

If the left hand side vector cannot be fully created, it will have zero length. The `out_of_memory` exception is thrown if enabled in the right hand side vector.

**Results:** The `operator =` public member function assigns the left hand side vector to be a copy of the right hand side.

**See Also:** `clear`, `clearAndDestroy`, `WExcept::out_of_memory`

## ***WPtrSortedVector<Type>,WPtrOrderedVector<Type>::operator ==( )***

---

**Synopsis:**

```
#include <wvector.h>
public:
int WPtrOrderedVector::operator ==( const WPtrOrderedVector
& ) const;
int WPtrSortedVector::operator ==( const WPtrSortedVector &
) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the class. Two vector objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side vectors are the same object. A FALSE (zero) value is returned otherwise.

**Synopsis:**

```
#include <wvector.h>
public:
int prepend( Type * );
```

**Semantics:** The `prepend` public member function inserts the passed element to be the first element in the vector. All vector elements contained in the vector are copied up one index.

This function is not provided by the `WCPtrSortedVector` class, since all elements must be inserted in sorted order by the `insert` member function.

Several different results can occur if the vector is not large enough for the new element. If the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the `prepend` fails if the amount the vector is to be grown (the second parameter to the constructor) is zero(0). Otherwise, the vector is automatically grown by the number of elements specified to the constructor, using the `resize` member function. If `resize` fails, the element is not inserted to the vector and the `out_of_memory` exception is thrown, if enabled.

**Results:** The `prepend` public member function prepends an element to the `WCPtrOrderedVector` object. A `TRUE` (non-zero) value is returned if the insert is successful. If the insert fails, a `FALSE` (zero) value is returned.

**See Also:** `append`, `insert`, `insertAt`, `WCEexcept::out_of_memory`, `WCEexcept::resize_required`

## ***WPtrSortedVector<Type>::remove(), WPtrOrderedVector<Type>::remove()***

---

**Synopsis:**

```
#include <wvector.h>
public:
Type * remove( const Type * );
```

**Semantics:** The `remove` public member function removes the first element in the vector which is equivalent to the passed value. Note that comparisons are done on the objects pointed to, not the pointers themselves. All vector elements stored after the removed elements are copied down one index.

A linear search is used by the `WPtrOrderedVector` class to find the element being removed. The `WPtrSortedVector` class uses a binary search.

**Results:** The `remove` public member function removes the first element in the vector which is equivalent to the passed value. The removed pointer is returned. If the vector did not contain an equivalent value, `NULL(0)` is returned.

**See Also:** `clear`, `clearAndDestroy`, `find`, `removeAll`, `removeAt`, `removeFirst`, `removeLast`



**Synopsis:**

```
#include <wvector.h>
public:
unsigned removeAll( const Type * );
```

**Semantics:** The `removeAll` public member function removes all elements in the vector which are equivalent to the passed value. Note that comparisons are done on the objects pointed to, not the pointers themselves. All vector elements stored after the removed elements are copied down one or more indexes to take the place of the removed elements.

A linear search is used by the `WPtrOrderedVector` class to find the elements being removed. The `WPtrSortedVector` class uses a binary search.

**Results:** The `removeAll` public member function removes all elements in the vector which are equivalent to the passed value. The number of elements removed is returned.

**See Also:** `clear`, `clearAndDestroy`, `find`, `occurrencesOf`, `remove`, `removeAt`, `removeFirst`, `removeLast`

## ***WCPtrSortedVector<Type>, WCPtrOrderedVector<Type>::removeAt()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
Type * removeAt( int );
```

**Semantics:** The `removeAt` public member function removes the element at the given index. All vector elements stored after the removed elements are copied down one index.

If the vector is empty and the `empty_container` exception is enabled, the exception is thrown.

If an attempt to remove an element with a negative index is made and the `index_range` exception is enabled, the exception is thrown. If the exception is not enabled, the first element is removed from the vector. Attempting to remove an element with index greater or equal to the number of entries in the vector also causes the `index_range` exception to be thrown if enabled. The last element in the vector is removed if the exception is not enabled.

**Results:** The `removeAt` public member function removes the element with the given index. If the index is invalid, the closest element to the given index is removed. The removed pointer is returned. If the vector was empty, `NULL(0)` is returned.

**See Also:** `clear`, `clearAndDestroy`, `insertAt`, `operator [ ]`, `remove`, `removeAll`, `removeFirst`, `removeLast`

**Synopsis:**

```
#include <wvector.h>
public:
Type * removeFirst();
```

**Semantics:** The `removeFirst` public member function removes the first element from a vector. All other vector elements are copied down one index.

If the vector is empty and the `empty_container` exception is enabled, the exception is thrown.

**Results:** The `removeFirst` public member function removes the first element from the vector. The removed pointer is returned. If the vector was empty, `NULL(0)` is returned.

**See Also:** `clear`, `clearAndDestroy`, `first`, `remove`, `removeAt`, `removeAll`, `removeLast`

## *WPtrSortedVector<Type>, WPtrOrderedVector<Type>::removeLast()*

---

**Synopsis:** `#include <wvector.h>`  
`public:`  
`Type * removeLast();`

**Semantics:** The `removeLast` public member function removes the last element from a vector. If the vector is empty and the `empty_container` exception is enabled, the exception is thrown.

**Results:** The `removeLast` public member function removes the last element from the vector. The removed pointer is returned. If the vector was empty, `NULL(0)` is returned.

**See Also:** `clear`, `clearAndDestroy`, `last`, `remove`, `removeAt`, `removeAll`, `removeFirst`

## ***WCPtrSortedVector<Type>::resize(), WCPtrOrderedVector<Type>::resize()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int resize( size_t new_size );
```

**Semantics:** The `resize` public member function is used to change the vector size to be able to store *new\_size* elements. If *new\_size* is larger than the previous vector size, all elements are copied into the newly sized vector, and new elements can be added using the `append`, `insert`, `insertAt`, and `prepend` member functions. If the vector is resized to a smaller size, the first *new\_size* elements are copied (all vector elements if the vector contained *new\_size* or fewer elements). The objects pointed to by the remaining elements are not deleted.

If the `resize` cannot be performed and the `out_of_memory` exception is enabled, the exception is thrown.

**Results:** The vector is resized to *new\_size*. A `TRUE` value (non-zero) is returned if the `resize` is successful. A `FALSE` (zero) result is returned if the `resize` fails.

**See Also:** `WCExcept::out_of_memory`

## ***WCPtrVector<Type>***

---

**Declared:** `wcvector.h`

The `WCPtrVector<Type>` class is a templated class used to store objects in a vector. Vectors are similar to arrays, but vectors perform bounds checking and can be resized. Elements are inserted into the vector by assigning to a vector index.

The `WCPtrOrderedVector` and `WCPtrSortedVector` classes are also available. They provide a more abstract view of the vector and additional functionality, including finding and removing elements.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type pointed to by the pointers stored in the vector.

The `WCEexcept` class is a base class of the `WCPtrVector<Type>` class and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCPtrVector<Type>` object. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Requirements of Type**

The `WCPtrVector<Type>` class requires nothing from `Type`.

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCPtrVector( size_t = 0 );
WCPtrVector( size_t, const Type * );
WCPtrVector( const WCPtrVector & );
virtual ~WCPtrVector();
void clear();
void clearAndDestroy();
size_t length() const;
int resize( size_t );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
Type * & operator [] ( int );
Type * const & operator [] ( int ) const;
WCPtrVector & operator =( const WCPtrVector & );
int operator ==( const WCPtrVector & ) const;
```

**Synopsis:**

```
#include <wvector.h>
public:
WPtrVector( size_t = 0 );
```

**Semantics:** The public `WPtrVector<Type>` constructor creates a `WPtrVector<Type>` object able to store the number of elements specified in the optional parameter, which defaults to zero. All vector elements are initialized to `NULL(0)`.

If the vector object cannot be fully initialized, the vector is created with length zero.

**Results:** The public `WPtrVector<Type>` constructor creates an initialized `WPtrVector<Type>` object with the specified length.

**See Also:** `WPtrVector<Type>`, `~WPtrVector<Type>`

## ***WPtrVector<Type>::WPtrVector()***

---

**Synopsis:**

```
#include <wvector.h>
public:
WPtrVector( size_t, const Type * );
```

**Semantics:** The public `WPtrVector<Type>` constructor creates a `WPtrVector<Type>` object able to store the number of elements specified by the first parameter. All vector elements are initialized to the pointer value given by the second parameter.

If the vector object cannot be fully initialized, the vector is created with length zero.

**Results:** The public `WPtrVector<Type>` constructor creates an initialized `WPtrVector<Type>` object with the specified length and elements set to the given value.

**See Also:** `WPtrVector<Type>`, `~WPtrVector<Type>`



**Synopsis:**

```
#include <wvector.h>
public:
WPtrVector( const WPtrVector & );
```

**Semantics:** The public `WPtrVector<Type>` constructor is the copy constructor for the `WPtrVector<Type>` class. The new vector is created with the same length as the given vector. All of the vector elements and exception trap states are copied.

If the new vector cannot be fully created, it will have length zero. The `out_of_memory` exception is thrown if enabled in the vector being copied.

**Results:** The public `WPtrVector<Type>` constructor creates a `WPtrVector<Type>` object which is a copy of the passed vector.

**See Also:** `operator =`, `WExcept::out_of_memory`

## ***WPtrVector<Type>::~WPtrVector()***

---

**Synopsis:**

```
#include <wvector.h>
public:
virtual ~WPtrVector();
```

**Semantics:** The public `~WPtrVector<Type>` destructor is the destructor for the `WPtrVector<Type>` class. If the vector is not length zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the vector elements are cleared using the `clear` member function. The objects which the vector elements point to are not deleted unless the `clearAndDestroy` member function is explicitly called before the destructor is called. The call to the public `~WPtrVector<Type>` destructor is inserted implicitly by the compiler at the point where the `WPtrVector<Type>` object goes out of scope.

**Results:** The public `~WPtrVector<Type>` destructor destroys an `WPtrVector<Type>` object.

**See Also:** `clear`, `clearAndDestroy`, `WExcept::not_empty`

**Synopsis:**

```
#include <wvector.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the vector so that it is of zero length. Objects pointed to by the vector elements are not deleted. The vector object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the vector to have zero length and no vector elements.

**See Also:** `~WPtrVector<Type>`, `clearAndDestroy`, `operator =`

## ***WCPtrVector<Type>::clearAndDestroy()***

---

**Synopsis:**

```
#include <wvector.h>
public:
void clearAndDestroy();
```

**Semantics:** The `clearAndDestroy` public member function is used to clear the vector to have zero length and delete the objects pointed to by the vector elements. The vector object is not destroyed and re-created by this function, so the vector object destructor is not invoked.

**Results:** The `clearAndDestroy` public member function clears the vector by deleting the objects pointed to by the vector elements and makes the vector zero length.

**See Also:** `clear`

**Synopsis:**

```
#include <wvector.h>
public:
size_t length() const;
```

**Semantics:** The `length` public member function is used to find the number of elements which can be stored in the `WPtrVector<Type>` object.

**Results:** The `length` public member function returns the length of the vector.

**See Also:** `resize`

## ***WCPtrVector<Type>::operator []()***

---

**Synopsis:**

```
#include <wvector.h>
public:
Type * & operator [] ( int );
Type * const & operator [] ( int ) const;
```

**Semantics:** `operator []` is the vector index operator. A reference to the object stored in the vector at the given index is returned. If a constant vector is indexed, a reference to a constant element is returned. The index operator of a non-constant vector is the only way to insert an element into the vector.

If an attempt to access an element with index greater than or equal to the length of a non-constant vector is made and the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the vector is automatically resized using the `resize` member function to have length the index value plus one. New vector elements are initialized to `NULL(0)`. If the resize failed, and the `out_of_memory` exception is enabled, the exception is thrown. If the exception is not enabled and the resize failed, the last element is indexed (a new element if the vector was zero length). If a negative value is used to index the non-constant vector and the `index_range` exception is enabled, the exception is thrown. If the exception is not enabled and the vector is empty, the `resize_required` exception may be thrown.

An attempt to index an empty constant vector may cause one of two exceptions to be thrown. If the `empty_container` exception is enabled, it is thrown. Otherwise, the `index_range` exception is thrown, if enabled. If neither exception is enabled, a first vector element is added and indexed (so that a reference to a valid element can be returned).

Indexing with a negative value or a value greater than or equal to the length of a constant vector causes the `index_range` exception to be thrown, if enabled.

**Results:** The `operator []` public member function returns a reference to the element at the given index. If the index is invalid, a reference to the closest valid element is returned. The result of the non-constant index operator may be assigned to.

**See Also:** `resize`, `WCEexcept::empty_container`, `WCEexcept::index_range`, `WCEexcept::out_of_memory`, `WCEexcept::resize_required`

**Synopsis:**

```
#include <wvector.h>
public:
WCPtrVector & operator =( const WCPtrVector & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCPtrVector<Type>` class. The left hand side vector is first cleared using the `clear` member function, and then the right hand side vector is copied. The left hand side vector is made to have the same length as the right hand side. All of the vector elements and exception trap states are copied.

If the left hand side vector cannot be fully created, it will have zero length. The `out_of_memory` exception is thrown if enabled in the right hand side vector.

**Results:** The `operator =` public member function assigns the left hand side vector to be a copy of the right hand side.

**See Also:** `clear`, `clearAndDestroy`, `WCEXCEPT::out_of_memory`

## ***WCPtrVector<Type>::operator ==()***

---

**Synopsis:**

```
#include <wvector.h>
public:
int operator ==( const WCPtrVector & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCPtrVector<Type>` class. Two vector objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side vectors are the same object. A FALSE (zero) value is returned otherwise.



**Synopsis:**

```
#include <wcvector.h>
public:
int resize( size_t new_size );
```

**Semantics:** The `resize` public member function is used to change the vector size to be able to store `new_size` elements. If `new_size` is larger than the previous vector size, all elements will be copied into the newly sized vector, and new elements are initialized to `NULL(0)`. If the vector is resized to a smaller size, the first `new_size` elements are copied. The objects pointed to by the remaining elements are not deleted.

If the `resize` cannot be performed and the `out_of_memory` exception is enabled, the exception is thrown.

**Results:** The vector is resized to `new_size`. A `TRUE` value (non-zero) is returned if the `resize` is successful. A `FALSE` (zero) result is returned if the `resize` fails.

**See Also:** `WCExcept::out_of_memory`

## *WCValSortedVector<Type>, WCValOrderedVector<Type>*

---

**Declared:** `wcvector.h`

The `WCValSortedVector<Type>` and `WCValOrderedVector<Type>` classes are templated classes used to store objects in a vector. Ordered and Sorted vectors are powerful arrays which can be resized and provide an abstract interface to insert, find and remove elements. An ordered vector maintains the order in which elements are added, and allows more than one copy of an element that is equivalent. The sorted vector allow only one copy of an equivalent element, and inserts them in a sorted order. The sorted vector is less efficient when inserting elements, but can provide a faster retrieval time.

Elements cannot be inserted into these vectors by assigning to a vector index. Vectors automatically grow when necessary to insert an element if the `resize_required` exception is not enabled.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type of the elements stored in the vector.

Values are copied into the vector, which could be undesirable if the stored objects are complicated and copying is expensive. Value vectors should not be used to store objects of a base class if any derived types of different sizes would be stored in the vector, or if the destructor for a derived class must be called.

The `WCValOrderedVector` class stores elements in the order which they are inserted using the `insert`, `append`, `prepend` and `insertAt` member functions. Linear searches are performed to locate entries, and the less than operator is not required.

The `WCValSortedVector` class stores elements in ascending order. This requires that `Type` provides a less than operator. Insertions are more expensive than inserting or appending into an ordered vector, since entries must be moved to make room for the new element. A binary search is used to locate elements in a sorted vector, making searches quicker than in the ordered vector.

Care must be taken when using the `WCValSortedVector` class not to change the ordering of the vector elements. The result returned by the index operator must not be assigned to or modified in such a way that it is no longer equivalent to the value inserted into the vector. Lookups assume elements are in sorted order.

The `WCValVector` class is also available. It provides a resizable and boundary safe vector similar to standard arrays.

The `WCExcept` class is a base class of the `WCValSortedVector<Type>` and `WCValOrderedVector<Type>` classes and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the

WCValSortedVector<Type> and WCValOrderedVector<Type> objects. No exceptions are enabled unless they are set by the exceptions member function.

### **Requirements of Type**

Both the WCValSortedVector<Type> and WCValOrderedVector<Type> classes require Type to have:

A default constructor ( Type::Type() ).

A well defined copy constructor ( Type::Type( const Type & ) ).

A well defined assignment operator  
( Type & operator =( const Type & ) ).

The following override of operator new() if Type overrides the global operator new():

```
void * operator new( size_t, void *ptr ) { return( ptr ); }
```

A well defined equivalence operator with constant parameters  
( int operator ==( const Type & ) const ).

Additionally the WCValSortedVector class requires Type to have:

A well defined less than operator with constant parameters  
( int operator <( const Type & ) const ).

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCValOrderedVector( size_t = WCDEFAULT_VECTOR_LENGTH, unsigned  
= WCDEFAULT_VECTOR_RESIZE_GROW );  
WCValOrderedVector( const WCValOrderedVector & );  
virtual ~WCValOrderedVector();  
WCValSortedVector( size_t = WCDEFAULT_VECTOR_LENGTH, unsigned  
= WCDEFAULT_VECTOR_RESIZE_GROW );  
WCValSortedVector( const WCValSortedVector & );  
virtual ~WCValSortedVector();  
void clear();  
int contains( const Type & ) const;  
unsigned entries() const;  
int find( const Type &, Type & ) const;  
Type first() const;
```

```
int index( const Type & ) const;
int insert( const Type & );
int isEmpty() const;
Type last() const;
int occurrencesOf( const Type & ) const;
int remove( const Type & );
unsigned removeAll( const Type & );
int removeAt( int );
int removeFirst();
int removeLast();
int resize( size_t );
```

The following public member functions are available for the `WCValOrderedVector` class only:

```
int append( const Type & );
int insertAt( int, const Type & );
int prepend( const Type & );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
Type & operator []( int );
const Type & operator []( int ) const;
WCValOrderedVector & WCValOrderedVector::operator =( const
WCValOrderedVector & );
WCValSortedVector & WCValSortedVector::operator =( const
WCValSortedVector & );
int WCValOrderedVector::operator ==( const WCValOrderedVector
& ) const;
int WCValSortedVector::operator ==( const WCValSortedVector &
) const;
```

**Synopsis:**

```
#include <wcvector.h>
public:
WCValOrderedVector( size_t = WCDEFAULT_VECTOR_LENGTH,
unsigned = WCDEFAULT_VECTOR_RESIZE_GROW );
```

**Semantics:** The `WCValOrderedVector<Type>` constructor creates an empty `WCValOrderedVector` object able to store the number of elements specified in the first optional parameter, which defaults to the constant `WCDEFAULT_VECTOR_LENGTH` (currently defined as 10). If the `resize_required` exception is not enabled, then the second optional parameter is used to specify the value to increase the vector size when an element is inserted into a full vector. If `zero(0)` is specified as the second parameter, any attempt to insert into a full vector fails. This parameter defaults to the constant `WCDEFAULT_VECTOR_RESIZE_GROW` (currently defined as 5).

If the vector object cannot be fully initialized, the vector is created with length zero.

**Results:** The `WCValOrderedVector<Type>` constructor creates an empty initialized `WCValOrderedVector` object.

**See Also:** `WCExcept::resize_required`

## ***WCValOrderedVector<Type>::WCValOrderedVector()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
WCValOrderedVector( const WCValOrderedVector & );
```

**Semantics:** The `WCValOrderedVector<Type>` constructor is the copy constructor for the `WCValOrderedVector` class. The new vector is created with the same length and `resize` value as the passed vector. All of the vector elements and exception trap states are copied.

If the new vector cannot be fully created, it will have length zero. The `out_of_memory` exception is thrown if enabled in the vector being copied.

**Results:** The `WCValOrderedVector<Type>` creates a `WCValOrderedVector` object which is a copy of the passed vector.

**See Also:** `operator =`, `WCExcept::out_of_memory`

**Synopsis:**

```
#include <wvector.h>
public:
virtual ~WCValOrderedVector();
```

**Semantics:** The `WCValOrderedVector<Type>` destructor is the destructor for the `WCValOrderedVector` class. If the vector is not length zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the vector entries are cleared using the `clear` member function. The call to the `WCValOrderedVector<Type>` destructor is inserted implicitly by the compiler at the point where the `WCValOrderedVector` object goes out of scope.

**Results:** The `WCValOrderedVector<Type>` destructor destroys an `WCValOrderedVector` object.

**See Also:** `clear`, `WCEXCEPT::not_empty`

## ***WCValSortedVector<Type>::WCValSortedVector()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
WCValSortedVector( size_t = WCDEFAULT_VECTOR_LENGTH,
unsigned = WCDEFAULT_VECTOR_RESIZE_GROW );
```

**Semantics:** The `WCValSortedVector<Type>` constructor creates an empty `WCValSortedVector` object able to store the number of elements specified in the first optional parameter, which defaults to the constant `WCDEFAULT_VECTOR_LENGTH` (currently defined as 10). If the `resize_required` exception is not enabled, then the second optional parameter is used to specify the value to increase the vector size when an element is inserted into a full vector. If zero(0) is specified as the second parameter, any attempt to insert into a full vector fails. This parameter defaults to the constant `WCDEFAULT_VECTOR_RESIZE_GROW` (currently defined as 5).

If the vector object cannot be fully initialized, the vector is created with length zero.

**Results:** The `WCValSortedVector<Type>` constructor creates an empty initialized `WCValSortedVector` object.

**See Also:** `WCExcept::resize_required`



**Synopsis:**

```
#include <wvector.h>
public:
WCValSortedVector( const WCValSortedVector & );
```

**Semantics:** The `WCValSortedVector<Type>` constructor is the copy constructor for the `WCValSortedVector` class. The new vector is created with the same length and resize value as the passed vector. All of the vector elements and exception trap states are copied.

If the new vector cannot be fully created, it will have length zero. The `out_of_memory` exception is thrown if enabled in the vector being copied.

**Results:** The `WCValSortedVector<Type>` constructor creates a `WCValSortedVector` object which is a copy of the passed vector.

**See Also:** `operator =`, `WCExcept::out_of_memory`

## ***WCValSortedVector<Type>::~~WCValSortedVector()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
virtual ~WCValSortedVector();
```

**Semantics:** The `WCValSortedVector<Type>` destructor is the destructor for the `WCValSortedVector` class. If the vector is not length zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the vector entries are cleared using the `clear` member function. The call to the `WCValSortedVector<Type>` destructor is inserted implicitly by the compiler at the point where the `WCValSortedVector` object goes out of scope.

**Results:** The `WCValSortedVector<Type>` destructor destroys an `WCValSortedVector` object.

**See Also:** `clear`, `WCExcept::not_empty`

**Synopsis:**

```
#include <wcvector.h>
public:
int append( const Type & );
```

**Semantics:** The `append` public member function appends the passed element to be the last element in the vector. The data stored in the vector is a copy of the data passed as a parameter. This member function has the same semantics as the `WCValOrderedVector::insert` member function.

This function is not provided by the `WCValSortedVector` class, since all elements must be inserted in sorted order by the `insert` member function.

Several different results can occur if the vector is not large enough for the new element. If the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the `append` fails if the amount the vector is to be grown (the second parameter to the constructor) is zero(0). Otherwise, the vector is automatically grown by the number of elements specified to the constructor, using the `resize` member function. If `resize` fails, the element is not appended to the vector and the `out_of_memory` exception is thrown, if enabled.

**Results:** The `append` public member function appends an element to the `WCValOrderedVector` object. A `TRUE` (non-zero) value is returned if the `append` is successful. If the `append` fails, a `FALSE` (zero) value is returned.

**See Also:** `insert`, `insertAt`, `prepend`, `WCExcept::out_of_memory`,  
`WCExcept::resize_required`

## ***WCValSortedVector<Type>::clear(), WCValOrderedVector<Type>::clear()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the vector so that it contains no entries, and is zero size. Elements stored in the vector are destroyed using `Type`'s destructor. The vector object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the vector to have zero length and no entries.

**See Also:** `~WCValOrderedVector`, `operator =`

## *WCValSortedVector<Type>,WCValOrderedVector<Type>::contains()*

---

**Synopsis:**

```
#include <wvector.h>
public:
int contains( const Type & ) const;
```

**Semantics:** The `contains` public member function is used to determine if a value is contained by a vector. A linear search is used by the `WCValOrderedVector` class to find the value. The `WCValSortedVector` class uses a binary search.

**Results:** The `contains` public member function returns a `TRUE` (non-zero) value if the element is found in the vector. A `FALSE` (zero) value is returned if the vector does not contain the element.

**See Also:** `index`, `find`

## ***WCValSortedVector<Type>::entries(), WCValOrderedVector<Type>::entries()***

---

**Synopsis:**

```
#include <wvector.h>
public:
    unsigned entries() const;
```

**Semantics:** The `entries` public member function is used to find the number of elements which are stored in the vector.

**Results:** The `entries` public member function returns the number of elements in the vector.

**See Also:** `isEmpty`

## ***WCValSortedVector<Type>::find(), WCValOrderedVector<Type>::find()***

---

**Synopsis:**

```
#include <wvector.h>
public:
int find( const Type &, Type & ) const;
```

**Semantics:** The `find` public member function is used to find an element equivalent to the first argument. The `WCValOrderedVector` class uses a linear search to find the element, and the `WCValSortedVector` class uses a binary search.

**Results:** If an equivalent element is found, a `TRUE` (non-zero) value is returned, and the second parameter is assigned the first equivalent value. A `FALSE` (zero) value is returned and the second parameter is unchanged if the element is not in the vector.

**See Also:** `contains`, `first`, `index`, `last`, `occurrencesOf`, `remove`

## ***WCValSortedVector<Type>::first(), WCValOrderedVector<Type>::first()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
Type first() const;
```

**Semantics:** The `first` public member function returns the first element in the vector. The element is not removed from the vector.

If the vector is empty, one of two exceptions can be thrown. The `empty_container` exception is thrown if it is enabled. Otherwise, if the `index_range` exception is enabled, it is thrown. If neither exception is enabled, a first element of the vector is added with a default value.

**Results:** The `first` public member function returns the value of the first element in the vector.

**See Also:** `last`, `removeFirst`, `WCEXCEPT::index_range`,  
`WCEXCEPT::resize_required`



## ***WCValSortedVector<Type>::index(), WCValOrderedVector<Type>::index()***

---

**Synopsis:**

```
#include <wvector.h>
public:
int index( const Type & ) const;
```

**Semantics:** The `index` public member function is used find the index of the first element equivalent to the passed element. A linear search is used by the `WCValOrderedVector` class to find the element. The `WCValSortedVector` class uses a binary search.

**Results:** The `index` public member function returns the index of the first element equivalent to the parameter. If the passed value is not contained in the vector, negative one (-1) is returned.

**See Also:** `contains`, `find`, `insertAt`, `operator []`, `removeAt`

## ***WCValSortedVector<Type>::insert(), WCValOrderedVector<Type>::insert()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int insert( const Type & );
```

**Semantics:** The `insert` public member function inserts the value into the vector. The data stored in the vector is a copy of the data passed as a parameter.

The `WCValOrderedVector::insert` function inserts the value as the last element of the vector, and has the same semantics as the `WCValOrderedVector::append` member function.

A binary search is performed to determine where the value should be inserted for the `WCValSortedVector::insert` function. Any elements greater than the inserted value are copied up one index (using `Type`'s assignment operator), so that the new element is after all elements with value less than or equal to it.

Several different results can occur if the vector is not large enough for the new element. If the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the `insert` fails if the amount the vector is to be grown (the second parameter to the constructor) is zero(0). Otherwise, the vector is automatically grown by the number of elements specified to the constructor, using the `resize` member function. If `resize` fails, the element is not inserted to the vector and the `out_of_memory` exception is thrown, if enabled.

**Results:** The `insert` public member function inserts an element in to the vector. A TRUE (non-zero) value is returned if the insert is successful. If the insert fails, a FALSE (zero) value is returned.

**See Also:** `append`, `insertAt`, `prepend`, `WCExcept::out_of_memory`, `WCExcept::resize_required`

**Synopsis:**

```
#include <wcvector.h>
public:
int insertAt( int, const Type & );
```

**Semantics:** The `insertAt` public member function inserts the second argument into the vector before the element at index given by the first argument. If the passed index is equal to the number of entries in the vector, the new value is appended to the vector as the last element. The data stored in the vector is a copy of the data passed as a parameter. All vector elements with indexes greater than or equal to the first parameter are copied (using `Type`'s assignment operator) up one index.

This function is not provided by the `WCValSortedVector` class, since all elements must be inserted in sorted order by the `insert` member function.

If the passed index is negative or greater than the number of entries in the vector and the `index_range` exception is enabled, the exception is thrown. If the exception is not enabled, the new element is inserted as the first element when the index is negative, or as the last element when the index is too large.

Several different results can occur if the vector is not large enough for the new element. If the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the insert fails if the amount the vector is to be grown (the second parameter to the constructor) is zero(0). Otherwise, the vector is automatically grown by the number of elements specified to the constructor, using the `resize` member function. If `resize` fails, the element is not inserted into the vector and the `out_of_memory` exception is thrown, if enabled.

**Results:** The `insertAt` public member function inserts an element into the `WCValOrderedVector` object before the element at the given index. A `TRUE` (non-zero) value is returned if the insert is successful. If the insert fails, a `FALSE` (zero) value is returned.

**See Also:** `append`, `insert`, `prepend`, `operator []`, `removeAt`, `WCExcept::index_range`, `WCExcept::out_of_memory`, `WCExcept::resize_required`

## ***WCValSortedVector<Type>,WCValOrderedVector<Type>::isEmpty()***

---

**Synopsis:**

```
#include <wvector.h>
public:
int isEmpty() const;
```

**Semantics:** The `isEmpty` public member function is used to determine if a vector object has any entries contained in it.

**Results:** A TRUE value (non-zero) is returned if the vector object does not have any vector elements contained within it. A FALSE (zero) result is returned if the vector contains at least one element.

**See Also:** `entries`

**Synopsis:**

```
#include <wvector.h>
public:
Type last() const;
```

**Semantics:** The `last` public member function returns the last element in the vector. The element is not removed from the vector.

If the vector is empty, one of two exceptions can be thrown. The `empty_container` exception is thrown if it is enabled. Otherwise, if the `index_range` exception is enabled, it is thrown. If neither exception is enabled, a first element of the vector is added with a default value.

**Results:** The `last` public member function returns the value of the last element in the vector.

**See Also:** `first`, `removeLast`, `WCEXCEPT::index_range`, `WCEXCEPT::resize_required`

## ***WCValSortedVector<Type>,WCValOrderedVector<Type>::occurrencesOf()***

---

**Synopsis:**

```
#include <wvector.h>
public:
int occurrencesOf( const Type & ) const;
```

**Semantics:** The `occurrencesOf` public member function returns the number of elements contained in the vector that are equivalent to the passed value. A linear search is used by the `WCValOrderedVector` class to find the value. The `WCValSortedVector` class uses a binary search.

**Results:** The `occurrencesOf` public member function returns the number of elements equivalent to the passed value.

**See Also:** `contains`, `find`, `index`, `operator []`, `removeAll`

**Synopsis:**

```
#include <wcvector.h>
public:
Type & operator [] ( int );
const Type & operator [] ( int ) const;
```

**Semantics:** `operator []` is the vector index operator. A reference to the object stored in the vector at the given index is returned. If a constant vector is indexed, a reference to a constant element is returned.

The `append`, `insert`, `insertAt` and `prepend` member functions are used to insert a new element into a vector, and the `remove`, `removeAll`, `removeAt`, `removeFirst` and `removeLast` member functions remove elements. The index operator cannot be used to change the number of entries in the vector. Searches may be performed using the `find` and `index` member functions.

If the vector is empty, one of two exceptions can be thrown. The `empty_container` exception is thrown if it is enabled. Otherwise, if the `index_range` exception is enabled, it is thrown. If neither exception is enabled, a first element of the vector is added with a default value. This element is added so that a reference to a valid vector element can be returned.

If the index value is negative and the `index_range` exception is enabled, the exception is thrown. An attempt to index an element with index greater than or equal to the number of entries in the vector will also cause the `index_range` exception to be thrown if enabled. If the exception is not enabled, attempting to index a negative element will index the first element in the vector, and attempting to index an element after the last entry will index the last element.

Care must be taken when using the `WCValSortedVector` class not to change the ordering of the vector elements. The result returned by the index operator must not be assigned to or modified in such a way that it is no longer equivalent (by `Type`'s equivalence operator) to the value inserted into the vector. Failure to comply may cause lookups to work incorrectly, since the binary search algorithm assumes elements are in sorted order.

**Results:** The `operator []` public member function returns a reference to the element at the given index. If the index is invalid, a reference to the closest valid element is returned. The result of the non-constant index operator may be assigned to.

**See Also:** `append`, `find`, `first`, `index`, `insert`, `insertAt`, `isEmpty`, `last`, `prepend`, `remove`, `removeAt`, `removeAll`, `removeFirst`, `removeLast`, `WCEXCEPT::empty_container`, `WCEXCEPT::index_range`

## ***WCValSortedVector<Type>,WCValOrderedVector<Type>::operator =()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
WCValOrderedVector & WCValOrderedVector::operator =( const
WCValOrderedVector & );
WCValSortedVector & WCValSortedVector::operator =( const
WCValSortedVector & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the class. The left hand side vector is first cleared using the `clear` member function, and then the right hand side vector is copied. The left hand side vector is made to have the same length and growth amount as the right hand side (the growth amount is the second argument passed to the right hand side vector constructor). All of the vector elements and exception trap states are copied.

If the left hand side vector cannot be fully created, it will have zero length. The `out_of_memory` exception is thrown if enabled in the right hand side vector.

**Results:** The `operator =` public member function assigns the left hand side vector to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`



## ***WCValSortedVector<Type>,WCValOrderedVector<Type>::operator ==( )***

---

**Synopsis:**

```
#include <wvector.h>
public:
int WCValOrderedVector::operator ==( const WCValOrderedVector
& ) const;
int WCValSortedVector::operator ==( const WCValSortedVector &
) const;
```

**Semantics:** The operator == public member function is the equivalence operator for the class. Two vector objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side vectors are the same object. A FALSE (zero) value is returned otherwise.

## ***WCValOrderedVector<Type>::prepend()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int prepend( const Type & );
```

**Semantics:** The `prepend` public member function inserts the passed element to be the first element in the vector. The data stored in the vector is a copy of the data passed as a parameter. All vector elements contained in the vector are copied (using `Type`'s assignment operator) up one index.

This function is not provided by the `WCValSortedVector` class, since all elements must be inserted in sorted order by the `insert` member function.

Several different results can occur if the vector is not large enough for the new element. If the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the `prepend` fails if the amount the vector is to be grown (the second parameter to the constructor) is zero(0). Otherwise, the vector is automatically grown by the number of elements specified to the constructor, using the `resize` member function. If `resize` fails, the element is not inserted to the vector and the `out_of_memory` exception is thrown, if enabled.

**Results:** The `prepend` public member function prepends an element to the `WCValOrderedVector` object. A `TRUE` (non-zero) value is returned if the insert is successful. If the insert fails, a `FALSE` (zero) value is returned.

**See Also:** `append`, `insert`, `insertAt`, `WCExcept::out_of_memory`, `WCExcept::resize_required`

## ***WCValSortedVector<Type>::remove(), WCValOrderedVector<Type>::remove()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int remove( const Type & );
```

**Semantics:** The `remove` public member function removes the first element in the vector which is equivalent to the passed value. All vector elements stored after the removed elements are copied (using `Type`'s assignment operator) down one index.

A linear search is used by the `WCValOrderedVector` class to find the element being removed. The `WCValSortedVector` class uses a binary search.

**Results:** The `remove` public member function removes the first element in the vector which is equivalent to the passed value. A `TRUE` (non-zero) value is returned if an equivalent element was contained in the vector and removed. If the vector did not contain an equivalent value, a `FALSE` (zero) value is returned.

**See Also:** `clear`, `find`, `removeAll`, `removeAt`, `removeFirst`, `removeLast`

## ***WCValSortedVector<Type>,WCValOrderedVector<Type>::removeAll()***

---

**Synopsis:**

```
#include <wvector.h>
public:
    unsigned removeAll( const Type & );
```

**Semantics:** The `removeAll` public member function removes all elements in the vector which are equivalent to the passed value. All vector elements stored after the removed elements are copied (using `Type`'s assignment operator) down one or more indexes to take the place of the removed elements.

A linear search is used by the `WCValOrderedVector` class to find the elements being removed. The `WCValSortedVector` class uses a binary search.

**Results:** The `removeAll` public member function removes all elements in the vector which are equivalent to the passed value. The number of elements removed is returned.

**See Also:** `clear`, `find`, `occurrencesOf`, `remove`, `removeAt`, `removeFirst`, `removeLast`

**Synopsis:**

```
#include <wcvector.h>
public:
int removeAt( int );
```

**Semantics:** The `removeAt` public member function removes the element at the given index. All vector elements stored after the removed elements are copied (using `Type`'s assignment operator) down one index.

If the vector is empty and the `empty_container` exception is enabled, the exception is thrown.

If an attempt to remove an element with a negative index is made and the `index_range` exception is enabled, the exception is thrown. If the exception is not enabled, the first element is removed from the vector. Attempting to remove an element with index greater or equal to the number of entries in the vector also causes the `index_range` exception to be thrown if enabled. The last element in the vector is removed if the exception is not enabled.

**Results:** The `removeAt` public member function removes the element with the given index. If the index is invalid, the closest element to the given index is removed. A `TRUE` (non-zero) value is returned if an element was removed. If the vector was empty, `FALSE` (zero) value is returned.

**See Also:** `clear`, `insertAt`, `operator []`, `remove`, `removeAll`, `removeFirst`, `removeLast`

## ***WCValSortedVector<Type>,WCValOrderedVector<Type>::removeFirst()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int removeFirst();
```

**Semantics:** The `removeFirst` public member function removes the first element from a vector. All other vector elements are copied (using `Type`'s assignment operator) down one index.

If the vector is empty and the `empty_container` exception is enabled, the exception is thrown.

**Results:** The `removeFirst` public member function removes the first element from the vector. A `TRUE` (non-zero) value is returned if an element was removed. If the vector was empty, `FALSE` (zero) value is returned.

**See Also:** `clear`, `first`, `remove`, `removeAt`, `removeAll`, `removeLast`

**Synopsis:**

```
#include <wcvector.h>
public:
int removeLast();
```

**Semantics:** The `removeLast` public member function removes the last element from a vector. If the vector is empty and the `empty_container` exception is enabled, the exception is thrown.

**Results:** The `removeLast` public member function removes the last element from the vector. A `TRUE` (non-zero) value is returned if an element was removed. If the vector was empty, `FALSE` (zero) value is returned.

**See Also:** `clear`, `last`, `remove`, `removeAt`, `removeAll`, `removeFirst`

## ***WCValSortedVector<Type>::resize(), WCValOrderedVector<Type>::resize()***

---

**Synopsis:**

```
#include <wcvector.h>
public:
int resize( size_t new_size );
```

**Semantics:** The `resize` public member function is used to change the vector size to be able to store `new_size` elements. If `new_size` is larger than the previous vector size, all elements are copied (using `Type`'s copy constructor) into the newly sized vector, and new elements can be added using the `append`, `insert`, `insertAt`, and `prepend` member functions. If the vector is resized to a smaller size, the first `new_size` elements are copied (all vector elements if the vector contained `new_size` or fewer elements). The remaining elements are destroyed using `Type`'s destructor.

If the `resize` cannot be performed and the `out_of_memory` exception is enabled, the exception is thrown.

**Results:** The vector is resized to `new_size`. A `TRUE` value (non-zero) is returned if the `resize` is successful. A `FALSE` (zero) result is returned if the `resize` fails.

**See Also:** `WCExcept::out_of_memory`



**Declared:** `wcvector.h`

The `WCValVector<Type>` class is a templated class used to store objects in a vector. Vectors are similar to arrays, but vectors perform bounds checking and can be resized. Elements are inserted into the vector by assigning to a vector index.

The `WCValOrderedVector` and `WCValSortedVector` classes are also available. They provide a more abstract view of the vector and additional functionality, including finding and removing elements.

Values are copied into the vector, which could be undesirable if the stored objects are complicated and copying is expensive. Value vectors should not be used to store objects of a base class if any derived types of different sizes would be stored in the vector, or if the destructor for a derived class must be called.

In the description of each member function, the text `Type` is used to indicate the template parameter defining the type of the elements stored in the vector.

The `WCExcept` class is a base class of the `WCValVector<Type>` class and provides the `exceptions` member function. This member function controls the exceptions which can be thrown by the `WCValVector<Type>` object. No exceptions are enabled unless they are set by the `exceptions` member function.

### **Requirements of Type**

The `WCValVector<Type>` class requires `Type` to have:

A default constructor (`Type::Type()`).

A well defined copy constructor (`Type::Type( const Type & )`).

The following override of `operator new()` only if `Type` overrides the global `operator new()`:

```
void * operator new( size_t, void *ptr ) { return( ptr ); }
```

### **Public Member Functions**

The following member functions are declared in the public interface:

```
WCValVector( size_t = 0 );  
WCValVector( size_t, const Type & );  
WCValVector( const WCValVector & );  
virtual ~WCValVector();  
void clear();
```

```
size_t length() const;  
int resize( size_t );
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
Type & operator [] ( int );  
const Type & operator [] ( int ) const;  
WCValVector & operator =( const WCValVector & );  
int operator ==( const WCValVector & ) const;
```

**Synopsis:**

```
#include <wvector.h>
public:
WCValVector( size_t = 0 );
```

**Semantics:** The public `WCValVector<Type>` constructor creates a `WCValVector<Type>` object able to store the number of elements specified in the optional parameter, which defaults to zero. All vector elements are initialized with `Type`'s default constructor.

If the vector object cannot be fully initialized, the vector is created with length zero.

**Results:** The public `WCValVector<Type>` constructor creates an initialized `WCValVector<Type>` object with the specified length.

**See Also:** `WCValVector<Type>`, `~WCValVector<Type>`

## ***WCValVector<Type>::WCValVector()***

---

**Synopsis:**

```
#include <wvector.h>
public:
WCValVector( size_t, const Type & );
```

**Semantics:** The public `WCValVector<Type>` constructor creates a `WCValVector<Type>` object able to store the number of elements specified by the first parameter. All vector elements are initialized to the value of the second parameter using `Type`'s copy constructor.

If the vector object cannot be fully initialized, the vector is created with length zero.

**Results:** The public `WCValVector<Type>` constructor creates an initialized `WCValVector<Type>` object with the specified length and elements set to the given value.

**See Also:** `WCValVector<Type>`, `~WCValVector<Type>`

**Synopsis:**

```
#include <wvector.h>
public:
WCValVector( const WCValVector & );
```

**Semantics:** The public `WCValVector<Type>` constructor is the copy constructor for the `WCValVector<Type>` class. The new vector is created with the same length as the given vector. All of the vector elements and exception trap states are copied.

If the new vector cannot be fully created, it will have length zero. The `out_of_memory` exception is thrown if enabled in the vector being copied.

**Results:** The public `WCValVector<Type>` constructor creates a `WCValVector<Type>` object which is a copy of the passed vector.

**See Also:** `operator =`, `WCExcept::out_of_memory`

## ***WCValVector<Type>::~~WCValVector()***

---

**Synopsis:**

```
#include <wvector.h>
public:
virtual ~WCValVector();
```

**Semantics:** The public `~WCValVector<Type>` destructor is the destructor for the `WCValVector<Type>` class. If the vector is not length zero and the `not_empty` exception is enabled, the exception is thrown. Otherwise, the vector elements are cleared using the `clear` member function. The call to the public `~WCValVector<Type>` destructor is inserted implicitly by the compiler at the point where the `WCValVector<Type>` object goes out of scope.

**Results:** The public `~WCValVector<Type>` destructor destroys an `WCValVector<Type>` object.

**See Also:** `clear`, `WCEXCEPT::not_empty`

**Synopsis:**

```
#include <wvector.h>
public:
void clear();
```

**Semantics:** The `clear` public member function is used to clear the vector so that it is of zero length. Elements stored in the vector are destroyed using `Type`'s destructor. The vector object is not destroyed and re-created by this function, so the object destructor is not invoked.

**Results:** The `clear` public member function clears the vector to have zero length and no vector elements.

**See Also:** `~WCValVector<Type>`, `operator =`

## ***WCVaVector<Type>::length()***

---

**Synopsis:**

```
#include <wvector.h>
public:
    size_t length() const;
```

**Semantics:** The `length` public member function is used to find the number of elements which can be stored in the `WCVaVector<Type>` object.

**Results:** The `length` public member function returns the length of the vector.

**See Also:** `resize`



**Synopsis:**

```
#include <wvector.h>
public:
Type & operator [] ( int );
const Type & operator [] ( int ) const;
```

**Semantics:** `operator []` is the vector index operator. A reference to the object stored in the vector at the given index is returned. If a constant vector is indexed, a reference to a constant element is returned. The index operator of a non-constant vector is the only way to insert an element into the vector.

If an attempt to access an element with index greater than or equal to the length of a non-constant vector is made and the `resize_required` exception is enabled, the exception is thrown. If the exception is not enabled, the vector is automatically resized using the `resize` member function to have length the index value plus one. New vector elements are initialized using `Type`'s default constructor. If the `resize` failed, and the `out_of_memory` exception is enabled, the exception is thrown. If the exception is not enabled and the `resize` failed, the last element is indexed (a new element if the vector was zero length). If a negative value is used to index the non-constant vector and the `index_range` exception is enabled, the exception is thrown. If the exception is not enabled and the vector is empty, the `resize_required` exception may be thrown.

An attempt to index an empty constant vector may cause one of two exceptions to be thrown. If the `empty_container` exception is enabled, it is thrown. Otherwise, the `index_range` exception is thrown, if enabled. If neither exception is enabled, a first vector element is added and indexed (so that a reference to a valid element can be returned).

Indexing with a negative value or a value greater than or equal to the length of a constant vector causes the `index_range` exception to be thrown, if enabled.

**Results:** The `operator []` public member function returns a reference to the element at the given index. If the index is invalid, a reference to the closest valid element is returned. The result of the non-constant index operator may be assigned to.

**See Also:** `resize`, `WCEexcept::empty_container`, `WCEexcept::index_range`, `WCEexcept::out_of_memory`, `WCEexcept::resize_required`

## ***WCValVector<Type>::operator =()***

---

**Synopsis:**

```
#include <wvector.h>
public:
WCValVector & operator =( const WCValVector & );
```

**Semantics:** The `operator =` public member function is the assignment operator for the `WCValVector<Type>` class. The left hand side vector is first cleared using the `clear` member function, and then the right hand side vector is copied. The left hand side vector is made to have the same length as the right hand side. All of the vector elements and exception trap states are copied.

If the left hand side vector cannot be fully created, it will have zero length. The `out_of_memory` exception is thrown if enabled in the right hand side vector.

**Results:** The `operator =` public member function assigns the left hand side vector to be a copy of the right hand side.

**See Also:** `clear`, `WCEXCEPT::out_of_memory`

**Synopsis:**

```
#include <wvector.h>
public:
int operator ==( const WCValVector & ) const;
```

**Semantics:** The `operator ==` public member function is the equivalence operator for the `WCValVector<Type>` class. Two vector objects are equivalent if they are the same object and share the same address.

**Results:** A TRUE (non-zero) value is returned if the left hand side and right hand side vectors are the same object. A FALSE (zero) value is returned otherwise.

## WCVaVector<Type>::resize()

---

**Synopsis:**

```
#include <wvector.h>
public:
int resize( size_t new_size );
```

**Semantics:** The `resize` public member function is used to change the vector size to be able to store `new_size` elements. If `new_size` is larger than the previous vector size, all elements will be copied (using `Type`'s copy constructor) into the newly sized vector, and new elements are initialized with `Type`'s default constructor. If the vector is resized to a smaller size, the first `new_size` elements are copied. The remaining elements are destroyed using `Type`'s destructor.

If the `resize` cannot be performed and the `out_of_memory` exception is enabled, the exception is thrown.

**Results:** The vector is resized to `new_size`. A `TRUE` value (non-zero) is returned if the `resize` is successful. A `FALSE` (zero) result is returned if the `resize` fails.

**See Also:** `WCEXCEPT::out_of_memory`

---

# ***18 Input/Output Classes***

The input/output stream classes provide program access to the file system. In addition, various options for formatting of output and reading of input are provided.

**Declared:** `fstream.h`

**Derived from:**

`streambuf`

The `filebuf` class is derived from the `streambuf` class, and provides additional functionality required to communicate with external files. Seek operations are supported when the underlying file supports seeking. Both input and output operations may be performed using a `filebuf` object, again when the underlying file supports read/write access.

`filebuf` objects are buffered by default, so the *reserve area* is allocated automatically unless one is specified when the `filebuf` object is created. The *get area* and *put area* pointers operate as if they were tied together. There is only one current position in a `filebuf` object.

The `filebuf` class allows only the *get area* or the *put area*, but not both, to be active at a time. This follows from the capability of files opened for both reading and writing to have operations of each type performed at arbitrary locations in the file. When writing is occurring, the characters are buffered in the *put area*. If a seek or read operation is done, the *put area* must be flushed before the next operation in order to ensure that the characters are written to the proper location in the file. Similarly, if reading is occurring, characters are buffered in the *get area*. If a write operation is done, the *get area* must be flushed and synchronized before the write operation in order to ensure the write occurs at the proper location in the file. If a seek operation is done, the *get area* does not have to be synchronized, but is discarded. When the *get area* is empty and a read is done, the `underflow` virtual member function reads more characters and fills the *get area* again. When the *put area* is full and a write is done, the `overflow` virtual member function writes the characters and makes the *put area* empty again.

C++ programmers who wish to use files without deriving new objects do not need to explicitly create or use a `filebuf` object.

### **Public Data Members**

The following data member is declared in the public interface. Its value is the default file protection that is used when creating new files. It is primarily referenced as a default argument in member functions.

```
static int const openprot;
```

### **Public Member Functions**

The following member functions are declared in the public interface:

```
filebuf();
filebuf( filedesc );
filebuf( filedesc, char *, int );
~filebuf();
int is_open() const;
filedesc fd() const;
filebuf *attach( filedesc );
filebuf *open( char const *,
ios::openmode,
int = filebuf::openprot );
filebuf *close();
virtual int pbackfail( int );
virtual int overflow( int = EOF );
virtual int underflow();
virtual streambuf *setbuf( char *, int );
virtual streampos seekoff( streamoff,
ios::seekdir,
ios::openmode );
virtual int sync();
```

**See Also:** `fstreambase`, `streambuf`

## ***filebuf::attach()***

---

**Synopsis:**

```
#include <fstream.h>
public:
filebuf *filebuf::attach( filedesc hdl );
```

**Semantics:** The `attach` public member function connects an existing `filebuf` object to an open file via the file's descriptor or handle specified by *hdl*. If the `filebuf` object is already connected to a file, the `attach` public member function fails. Otherwise, the `attach` public member function extracts information from the file system to determine the capabilities of the file and hence the `filebuf` object.

**Results:** The `attach` public member function returns a pointer to the `filebuf` object on success, otherwise `NULL` is returned.

**See Also:** `filebuf`, `fd`, `open`



**Synopsis:**

```
#include <fstream.h>
public:
filebuf *filebuf::close();
```

**Semantics:** The `close` public member function disconnects the `filebuf` object from a connected file and closes the file. Any buffered output is flushed before the file is closed.

**Results:** The `close` public member function returns a pointer to the `filebuf` object on success, otherwise `NULL` is returned.

**See Also:** `filebuf`, `fd`, `is_open`

## ***filebuf::fd()***

---

**Synopsis:**

```
#include <fstream.h>
public:
filedesc filebuf::fd() const;
```

**Semantics:** The `fd` public member function queries the state of the `filebuf` object file handle.

**Results:** The `fd` public member function returns the file descriptor or handle of the file to which the `filebuf` object is currently connected. If the `filebuf` object is not currently connected to a file, EOF is returned.

**See Also:** `filebuf::attach`, `is_open`

**Synopsis:**

```
#include <fstream.h>
public:
filebuf::filebuf();
```

**Semantics:** This form of the public `filebuf` constructor creates a `filebuf` object that is not currently connected to any file. A call to the `fd` member function for this created `filebuf` object returns `EOF`, unless a file is connected using the `attach` member function.

**Results:** The public `filebuf` constructor produces a `filebuf` object that is not currently connected to any file.

**See Also:** `~filebuf`, `attach`, `open`

## ***filebuf::filebuf()***

---

**Synopsis:**

```
#include <fstream.h>
public:
filebuf::filebuf( filedesc hdl );
```

**Semantics:** This form of the public `filebuf` constructor creates a `filebuf` object that is connected to an open file. The file is specified via the *hdl* parameter, which is a file descriptor or handle.

This form of the public `filebuf` constructor is similar to using the default constructor, and calling the `attach` member function. A call to the `fd` member function for this created `filebuf` object returns *hdl*.

**Results:** The public `filebuf` constructor produces a `filebuf` object that is connected to *hdl*.

**See Also:** `~filebuf`, `attach`, `open`

**Synopsis:**

```
#include <fstream.h>
public:
filebuf::filebuf( filedesc hdl, char *buf, int len );
```

**Semantics:** This form of the public `filebuf` constructor creates a `filebuf` object that is connected to an open file and that uses the buffer specified by *buf* and *len*. The file is specified via the *hdl* parameter, which is a file descriptor or handle. If *buf* is `NULL` and/or *len* is less than or equal to zero, the `filebuf` object is unbuffered, so that reading and/or writing take place one character at a time.

This form of the public `filebuf` constructor is similar to using the default constructor, and calling the `attach` and `setbuf` member functions.

**Results:** The public `filebuf` constructor produces a `filebuf` object that is connected to *hdl*.

**See Also:** `~filebuf`, `attach`, `open`, `setbuf`

## ***filebuf::~~filebuf()***

---

**Synopsis:**

```
#include <fstream.h>
public:
filebuf::~~filebuf();
```

**Semantics:** The public `~filebuf` destructor closes the file if it was explicitly opened using the `open` member function. Otherwise, the destructor takes no explicit action. The `streambuf` destructor is called to destroy that portion of the `filebuf` object. The call to the public `~filebuf` destructor is inserted implicitly by the compiler at the point where the `filebuf` object goes out of scope.

**Results:** The `filebuf` object is destroyed.

**See Also:** `~filebuf`, `close`

**Synopsis:**

```
#include <fstream.h>
public:
int filebuf::is_open();
```

**Semantics:** The `is_open` public member function queries the `filebuf` object state.

**Results:** The `is_open` public member function returns a non-zero value if the `filebuf` object is currently connected to a file. Otherwise, zero is returned.

**See Also:** `filebuf::attach`, `close`, `fd`, `open`

## ***filebuf::open()***

---

**Synopsis:**

```
#include <fstream.h>
public:
filebuf *filebuf::open( const char *name,
ios::openmode mode,
int prot = filebuf::openprot );
```

**Semantics:** The `open` public member function is used to connect the `filebuf` object to a file specified by the *name* parameter. The file is opened using the specified *mode*. For details about the *mode* parameter, see the description of `ios::openmode`. The *prot* parameter specifies the file protection attributes to use when creating a file.

**Results:** The `open` public member function returns a pointer to the `filebuf` object on success, otherwise `NULL` is returned.

**See Also:** `filebuf`, `close`, `is_open`, `openprot`



**Synopsis:**

```
#include <fstream.h>
public:
static int const filebuf::openprot;
```

**Semantics:** The `openprot` public member data is used to specify the default file protection to be used when creating new files. This value is used as the default if no user specified value is provided.

The default value is octal 0644. This is generally interpreted as follows:

- Owner: read/write
- Group: read
- World: read

Note that not all operating systems support all bits.

**See Also:** `filebuf`, `open`

## ***filebuf::overflow()***

---

**Synopsis:**

```
#include <fstream.h>
public:
virtual int filebuf::overflow( int ch = EOF );
```

**Semantics:** The `overflow` public virtual member function provides the output communication to the file to which the `filebuf` object is connected. Member functions in the `streambuf` class call the `overflow` public virtual member function for the derived class when the *put area* is full.

The `overflow` public virtual member function performs the following steps:

1. If no buffer is present, a buffer is allocated with the `streambuf::allocate` member function, which may call the `doallocate` virtual member function. The *put area* is then set up. If, after calling `streambuf::allocate`, no buffer is present, the `filebuf` object is unbuffered and *ch* (if not `EOF`) is written directly to the file without buffering, and no further action is taken.
2. If the *get area* is present, it is flushed with a call to the `sync` virtual member function. Note that the *get area* won't be present if a buffer was set up in step 1.
3. If *ch* is not `EOF`, it is added to the *put area*, if possible.
4. Any characters in the *put area* are written to the file.
5. The *put area* pointers are updated to reflect the new state of the *put area*. If the write did not complete, the unwritten portion of the *put area* is still present. If the *put area* was full before the write, *ch* (if not `EOF`) is placed at the start of the *put area*. Otherwise, the *put area* is empty.

**Results:** The `overflow` public virtual member function returns `__NOT_EOF` on success, otherwise `EOF` is returned.

**See Also:** `streambuf::overflow`  
`filebuf::underflow`

**Synopsis:**

```
#include <fstream.h>
public:
virtual int filebuf::pbackfail( int ch );
```

**Semantics:** The `pbackfail` public virtual member function handles an attempt to put back a character when there is no room at the beginning of the *get area*. The `pbackfail` public virtual member function first calls the `sync` virtual member function to flush the *put area* and then it attempts to seek backwards over *ch* in the associated file.

**Results:** The `pbackfail` public virtual member function returns *ch* on success, otherwise EOF is returned.

**See Also:** `streambuf::pbackfail`

## ***filebuf::seekoff()***

---

**Synopsis:**

```
#include <fstream.h>
public:
virtual streampos filebuf::seekoff( streamoff offset,
ios::seekdir dir,
ios::openmode mode );
```

**Semantics:** The `seekoff` public virtual member function is used to position the `filebuf` object (and hence the file) to a particular offset so that subsequent input or output operations commence from that point. The offset is specified by the *offset* and *dir* parameters.

Since the *get area* and *put area* pointers are tied together for the `filebuf` object, the *mode* parameter is ignored.

Before the actual seek occurs, the *get area* and *put area* of the `filebuf` object are flushed via the `sync` virtual member function. Then, the new position in the file is calculated and the seek takes place.

The *dir* parameter may be `ios::beg`, `ios::cur`, or `ios::end` and is interpreted in conjunction with the *offset* parameter as follows:

`ios::beg` the *offset* is relative to the start and should be a positive value.  
`ios::cur` the *offset* is relative to the current position and may be positive (seek towards end) or negative (seek towards start).  
`ios::end` the *offset* is relative to the end and should be a negative value.

If the *dir* parameter has any other value, or the *offset* parameter does not have an appropriate sign, the `seekoff` public virtual member function fails.

**Results:** The `seekoff` public virtual member function returns the new position in the file on success, otherwise EOF is returned.

**See Also:** `streambuf::seekoff`

**Synopsis:**

```
#include <fstream.h>
public:
virtual ostreambuf *filebuf::setbuf( char *buf, int len );
```

**Semantics:** The `setbuf` public virtual member function is used to offer a buffer, specified by *buf* and *len* to the `filebuf` object. If the *buf* parameter is `NULL` or the *len* is less than or equal to zero, the request is to make the `filebuf` object unbuffered.

If the `filebuf` object is already connected to a file and has a buffer, the offer is rejected. In other words, a call to the `setbuf` public virtual member function after the `filebuf` object has started to be used usually fails because the `filebuf` object has set up a buffer.

If the request is to make the `filebuf` object unbuffered, the offer succeeds.

If the *buf* is too small (less than five characters), the offer is rejected. Five characters are required to support the default putback area.

Otherwise, the *buf* is acceptable and the offer succeeds.

If the offer succeeds, the `ostreambuf::setb` member function is called to set up the pointers to the buffer. The `ostreambuf::setb` member function releases the old buffer (if present), depending on how that buffer was allocated.

Calls to the `setbuf` public virtual member function are usually made by a class derived from the `fstream` class, not directly by a user program.

**Results:** The `setbuf` public virtual member function returns a pointer to the `filebuf` object on success, otherwise `NULL` is returned.

**See Also:** `ostreambuf::setbuf`

## ***filebuf::sync()***

---

**Synopsis:**

```
#include <fstream.h>
public:
virtual int filebuf::sync();
```

**Semantics:** The `sync` public virtual member function synchronizes the `filebuf` object with the external file or device. If the *put area* contains characters it is flushed. This leaves the file positioned after the last written character. If the *get area* contains buffered (unread) characters, file is backed up to be positioned after the last read character.

Note that the *get area* and *put area* never both contain characters.

**Results:** The `sync` public virtual member function returns `__NOT_EOF` on success, otherwise `EOF` is returned.

**See Also:** `streambuf::sync`

**Synopsis:**

```
#include <fstream.h>
public:
virtual int filebuf::underflow();
```

**Semantics:** The `underflow` public virtual member function provides the input communication from the file to which the `filebuf` object is connected. Member functions in the `streambuf` class call the `underflow` public virtual member function for the derived class when the *get area* is empty.

The `underflow` public virtual member function performs the following steps:

1. If no *reserve area* is present, a buffer is allocated with the `streambuf::allocate` member function, which may call the `doallocate` virtual member function. If, after calling `allocate`, no *reserve area* is present, the `filebuf` object is unbuffered and a one-character *reserve area* (plus *putback area*) is set up to do unbuffered input. This buffer is embedded in the `filebuf` object. The *get area* is set up as empty.
2. If the *put area* is present, it is flushed using the `sync` virtual member function.
3. The unused part of the *get area* is used to read characters from the file connected to the `filebuf` object. The *get area* pointers are then set up to reflect the new *get area*.

**Results:** The `underflow` public virtual member function returns the first unread character of the *get area*, on success, otherwise EOF is returned. Note that the *get pointer* is not advanced on success.

**See Also:** `streambuf::underflow`  
`filebuf::overflow`

**Declared:** fstream.h

**Derived from:**

fstreambase, iostream

The `fstream` class is used to access files for reading and writing. The file can be opened and closed, and read, write and seek operations can be performed.

The `fstream` class provides very little of its own functionality. It is derived from both the `fstreambase` and `iostream` classes. The `fstream` constructors, destructor and member function provide simplified access to the appropriate equivalents in the base classes.

Of the available I/O stream classes, creating an `fstream` object is the preferred method of accessing a file for both input and output.

### **Public Member Functions**

The following public member functions are declared:

```
fstream();  
fstream( char const *,  
ios::openmode = ios::in|ios::out,  
int = filebuf::openprot );  
fstream( filedesc );  
fstream( filedesc, char *, int );  
~fstream();  
void open( char const *,  
ios::openmode = ios::in|ios::out,  
int = filebuf::openprot );
```

**See Also:** fstreambase, ifstream, iostream, ofstream



**Synopsis:**

```
#include <fstream.h>
public:
fstream::fstream();
```

**Semantics:** This form of the public `fstream` constructor creates an `fstream` object that is not connected to a file. The `open` or `attach` member functions should be used to connect the `fstream` object to a file.

**Results:** The public `fstream` constructor produces an `fstream` object that is not connected to a file.

**See Also:** `~fstream`, `open`, `fstreambase::attach`

## ***fstream::fstream()***

---

**Synopsis:**

```
#include <fstream.h>
public:
fstream::fstream( const char *name,
ios::openmode mode = ios::in|ios::out,
int prot = filebuf::openprot );
```

**Semantics:** This form of the public `fstream` constructor creates an `fstream` object that is connected to the file specified by the *name* parameter, using the specified *mode* and *prot* parameters. The connection is made via the C library `open` function.

**Results:** The public `fstream` constructor produces an `fstream` object that is connected to the file specified by *name*. If the `open` fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `~fstream`, `open`, `openmode`, `openprot`

**Synopsis:**

```
#include <fstream.h>
public:
fstream::fstream( filedesc hdl );
```

**Semantics:** This form of the public `fstream` constructor creates an `fstream` object that is attached to the file specified by the *hdl* parameter.

**Results:** The public `fstream` constructor produces an `fstream` object that is attached to *hdl*. If the attach fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `~fstream`, `fstreambase::attach`, `fstreambase::fd`

## ***fstream::fstream()***

---

**Synopsis:**

```
#include <fstream.h>
public:
fstream::fstream( filedesc hdl, char *buf, int len );
```

**Semantics:** This form of the public `fstream` constructor creates an `fstream` object that is connected to the file specified by the `hdl` parameter. The buffer specified by the `buf` and `len` parameters is offered to the associated `filebuf` object via the `setbuf` member function. If the `buf` parameter is `NULL` or the `len` is less than or equal to zero, the `filebuf` is unbuffered, so that each read or write operation reads or writes a single character at a time.

**Results:** The public `fstream` constructor produces an `fstream` object that is attached to `hdl`. If the connection to `hdl` fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object. If the `setbuf` fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `~fstream`, `filebuf::setbuf`, `fstreambase::attach`

**Synopsis:** `#include <fstream.h>`  
`public:`  
`fstream::~~fstream();`

**Semantics:** The public `~fstream` destructor does not do anything explicit. The call to the public `~fstream` destructor is inserted implicitly by the compiler at the point where the `fstream` object goes out of scope.

**Results:** The public `~fstream` destructor destroys the `fstream` object.

**See Also:** `fstream`

## ***fstream::open()***

---

**Synopsis:**

```
#include <fstream.h>
public:
void fstream::open( const char *name,
ios::openmode mode = ios::in|ios::out,
int prot = filebuf::openprot );
```

**Semantics:** The open public member function connects the *fstream* object to the file specified by the *name* parameter, using the specified *mode* and *prot* parameters. The *mode* parameter is optional and usually is not specified unless additional bits (such as *ios::binary* or *ios::text*) are to be specified. The connection is made via the C library *open* function.

**Results:** If the open fails, *ios::failbit* is set in the error state in the inherited *ios* object.

**See Also:** *fstreambase::attach*, *fstreambase::close*, *fstreambase::fd*,  
*fstreambase::is\_open*  
*fstream::openmode*, *openprot*

**Declared:** fstream.h

**Derived from:**  
ios

**Derived by:** ifstream, ofstream, fstream

The `fstreambase` class is a base class that provides common functionality for the three file-based classes, `ifstream`, `ofstream` and `fstream`. The `fstreambase` class is derived from the `ios` class, providing the stream state information, plus it provides member functions for opening and closing files. The actual file manipulation work is performed by the `filebuf` class.

It is not intended that `fstreambase` objects be created. Instead, the user should create an `ifstream`, `ofstream` or `fstream` object.

### **Protected Member Functions**

The following member functions are declared in the protected interface:

```
fstreambase();  
fstreambase( char const *,  
ios::openmode,  
int = filebuf::openprot );  
fstreambase( filedesc );  
fstreambase( filedesc, char *, int );  
~fstreambase();
```

### **Public Member Functions**

The following member functions are declared in the public interface:

```
void attach( filedesc );  
void close();  
filedesc fd() const;  
int is_open() const;  
void open( char const *,  
ios::openmode,  
int = filebuf::openprot );  
filebuf *rdbuf() const;  
void setbuf( char *, int );
```

**See Also:** filebuf, fstream, ifstream, ofstream

## ***fstreambase::attach()***

---

**Synopsis:**

```
#include <fstream.h>
public:
void fstreambase::attach( filedesc hdl );
```

**Semantics:** The `attach` public member function connects the `fstreambase` object to the file specified by the `hdl` parameter.

**Results:** If the `attach` public member function fails, `ios::failbit` bit is set in the error state in the inherited `ios` object. The error state in the inherited `ios` object is cleared on success.

**See Also:** `fstreambase::fd`, `is_open`, `open`



**Synopsis:**

```
#include <fstream.h>
public:
void fstreambase::close();
```

**Semantics:** The `close` public member function disconnects the `fstreambase` object from the file with which it is associated. If the `fstreambase` object is not associated with a file, the `close` public member function fails.

**Results:** If the `close` public member function fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `fstreambase::fd`, `is_open`, `open`

## ***fstreambase::fstreambase()***

---

**Synopsis:** `#include <fstream.h>`  
`protected:`  
`fstreambase::fstreambase();`

**Semantics:** The protected `fstreambase` constructor creates an `fstreambase` object that is initialized, but not connected to anything. The `open` or `attach` member function should be used to connect the `fstreambase` object to a file.

**Results:** The protected `fstreambase` constructor produces an `fstreambase` object.

**See Also:** `~fstreambase`, `attach`, `open`

**Synopsis:**

```
#include <fstream.h>
protected:
fstreambase::fstreambase( char const *name,
ios::openmode mode,
int prot = filebuf::openprot );
```

**Semantics:** This protected `fstreambase` constructor creates an `fstreambase` object that is initialized and connected to the file indicated by *name* using the specified *mode* and *prot*. The `fstreambase` object is connected to the specified file via the open C library function.

**Results:** The protected `fstreambase` constructor produces an `fstreambase` object. If the call to open for the file fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `~fstreambase`, `open`, `openmode`, `openprot`

## ***fstreambase::fstreambase()***

---

**Synopsis:** `#include <fstream.h>`  
`protected:`  
`fstreambase::fstreambase( filedesc hdl );`

**Semantics:** This protected `fstreambase` constructor creates an `fstreambase` object that is initialized and connected to the open file specified by the *hdl* parameter.

**Results:** The protected `fstreambase` constructor produces an `fstreambase` object. If the attach to the file fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `~fstreambase`, `attach`

**Synopsis:** `#include <fstream.h>`  
`protected:`  
`fstreambase::fstreambase( filedesc hdl, char *buf, int len );`

**Semantics:** This protected `fstreambase` constructor creates an `fstreambase` object that is initialized and connected to the open file specified by the `hdl` parameter. The buffer, specified by the `buf` and `len` parameters, is offered via the `setbuf` virtual member function to be used as the *reserve area* for the `filebuf` associated with the `fstreambase` object.

**Results:** The protected `fstreambase` constructor produces an `fstreambase` object. If the attach to the file fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `~fstreambase`, `attach`, `setbuf`

## ***fstreambase::~~fstreambase()***

---

**Synopsis:** `#include <fstream.h>`  
`protected:`  
`fstreambase::~~fstreambase();`

**Semantics:** The protected `~fstreambase` destructor does not do anything explicit. The `filebuf` object associated with the `fstreambase` object is embedded within the `fstreambase` object, so the `filebuf` destructor is called. The `ios` destructor is called for that portion of the `fstreambase` object. The call to the protected `~fstreambase` destructor is inserted implicitly by the compiler at the point where the `fstreambase` object goes out of scope.

**Results:** The `fstreambase` object is destroyed.

**See Also:** `fstreambase`, `close`

**Synopsis:**

```
#include <fstream.h>
public:
int fstreambase::is_open() const;
```

**Semantics:** The `is_open` public member function queries the current state of the file associated with the `fstreambase` object. Calling the `is_open` public member function is equivalent to calling the `fd` member function and testing for EOF.

**Results:** The `is_open` public member function returns a non-zero value if the `fstreambase` object is currently connected to a file, otherwise zero is returned.

**See Also:** `fstreambase::attach`, `fd`, `open`

## ***fstreambase::fd()***

---

**Synopsis:**

```
#include <fstream.h>
public:
filedesc fstreambase::fd() const;
```

**Semantics:** The `fd` public member function returns the file descriptor for the file to which the `fstreambase` object is connected.

**Results:** The `fd` public member function returns the file descriptor for the file to which the `fstreambase` object is connected. If the `fstreambase` object is not currently connected to a file, EOF is returned.

**See Also:** `fstreambase::attach`, `is_open`, `open`



**Synopsis:**

```
#include <fstream.h>
public:
void fstreambase::open( const char *name,
ios::openmode mode,
int prot = filebuf::openprot );
```

**Semantics:** The open public member function connects the fstreambase object to the file specified by *name*, using the specified *mode* and *prot*. The connection is made via the C library open function.

**Results:** If the open fails, ios::failbit is set in the error state in the inherited ios object. The error state in the inherited ios object is cleared on success.

**See Also:** fstreambase::attach, close, fd, is\_open, openmode, openprot

## ***fstreambase::rdbuf()***

---

**Synopsis:**

```
#include <fstream.h>
public:
filebuf *fstreambase::rdbuf() const;
```

**Semantics:** The `rdbuf` public member function returns the address of the `filebuf` object currently associated with the `fstreambase` object.

**Results:** The `rdbuf` public member function returns a pointer to the `filebuf` object currently associated with the `fstreambase` object. If there is no associated `filebuf`, `NULL` is returned.

**See Also:** `ios::rdbuf`

**Synopsis:**

```
#include <fstream.h>
public:
void fstreambase::setbuf( char *buf, int len );
```

**Semantics:** The `setbuf` public member function offers the specified buffer to the `filebuf` object associated with the `fstreambase` object. The `filebuf` may or may not reject the offer, depending upon its state.

**Results:** If the offer is rejected, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `filebuf::setbuf`

**Declared:** `fstream.h`

**Derived from:**

`fstreambase`, `istream`

The `ifstream` class is used to access existing files for reading. Such files can be opened and closed, and read and seek operations can be performed.

The `ifstream` class provides very little of its own functionality. Derived from both the `fstreambase` and `istream` classes, its constructors, destructor and member functions provide simplified access to the appropriate equivalents in those base classes.

Of the available I/O stream classes, creating an `ifstream` object is the preferred method of accessing a file for input only operations.

### **Public Member Functions**

The following public member functions are declared:

```
ifstream();
ifstream( char const *,
ios::openmode = ios::in,
int = filebuf::openprot );
ifstream( filedesc );
ifstream( filedesc, char *, int );
~ifstream();
void open( char const *,
ios::openmode = ios::in,
int = filebuf::openprot );
```

**See Also:** `fstream`, `fstreambase`, `istream`, `ofstream`

**Synopsis:**

```
#include <fstream.h>
public:
ifstream::ifstream();
```

**Semantics:** This form of the public `ifstream` constructor creates an `ifstream` object that is not connected to a file. The `open` or `attach` member functions should be used to connect the `ifstream` object to a file.

**Results:** The public `ifstream` constructor produces an `ifstream` object that is not connected to a file.

**See Also:** `~ifstream`, `open`, `fstreambase::attach`

## ***ifstream::ifstream()***

---

**Synopsis:**

```
#include <fstream.h>
public:
ifstream::ifstream( const char *name,
ios::openmode mode = ios::in,
int prot = filebuf::openprot );
```

**Semantics:** This form of the public `ifstream` constructor creates an `ifstream` object that is connected to the file specified by the *name* parameter, using the specified *mode* and *prot* parameters. The connection is made via the C library `open` function.

**Results:** The public `ifstream` constructor produces an `ifstream` object that is connected to the file specified by *name*. If the `open` fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `~ifstream`, `open`, `openmode`, `openprot`, `fstreambase::attach`, `fstreambase::fd`, `fstreambase::is_open`

**Synopsis:**

```
#include <fstream.h>
public:
ifstream::ifstream( filedesc hdl );
```

**Semantics:** This form of the public `ifstream` constructor creates an `ifstream` object that is attached to the file specified by the *hdl* parameter.

**Results:** The public `ifstream` constructor produces an `ifstream` object that is attached to *hdl*. If the attach fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `fstreambase::attach`  
`~ifstream`, `open`

## ***ifstream::ifstream()***

---

**Synopsis:**

```
#include <fstream.h>
public:
ifstream::ifstream( filedesc hdl, char *buf, int len );
```

**Semantics:** This form of the public `ifstream` constructor creates an `ifstream` object that is connected to the file specified by the `hdl` parameter. The buffer specified by the `buf` and `len` parameters is offered to the associated `filebuf` object via the `setbuf` member function. If the `buf` parameter is `NULL` or the `len` is less than or equal to zero, the `filebuf` is unbuffered, so that each read or write operation reads or writes a single character at a time.

**Results:** The public `ifstream` constructor produces an `ifstream` object that is attached to `hdl`. If the connection to `hdl` fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object. If the `setbuf` fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `fstreambase::attach`, `fstreambase::setbuf`  
`~ifstream`, `open`



**Synopsis:**

```
#include <fstream.h>
public:
ifstream::~ifstream();
```

**Semantics:** The public `~ifstream` destructor does not do anything explicit. The call to the public `~ifstream` destructor is inserted implicitly by the compiler at the point where the `ifstream` object goes out of scope.

**Results:** The public `~ifstream` destructor destroys the `ifstream` object.

**See Also:** `ifstream`

## ***ifstream::open()***

---

**Synopsis:**

```
#include <fstream.h>
public:
void ifstream::open( const char *name,
ios::openmode mode = ios::in,
int prot = filebuf::openprot );
```

**Semantics:** The open public member function connects the ifstream object to the file specified by the *name* parameter, using the specified *mode* and *prot* parameters. The mode parameter is optional and usually is not specified unless additional bits (such as ios::binary or ios::text) are to be specified. The connection is made via the C library open function.

**Results:** If the open fails, ios::failbit is set in the error state in the inherited ios object.

**See Also:** fstreambase::attach, fstreambase::close, fstreambase::fd,  
fstreambase::is\_open  
ifstream::openmode, openprot

---

**Declared:** `iostream.h`

**Derived by:** `istream, ostream`

The `ios` class is used to group together common functionality needed for other derived stream classes. It is not intended that objects of type `ios` be created.

This class maintains state information about the stream. (the `ios` name can be thought of as a short-form for I/O State). Error flags, formatting flags, and values and the connection to the buffers used for the input and output are all maintained by the `ios` class. No information about the buffer itself is stored in an `ios` object, merely the pointer to the buffer information.

### Protected Member Functions

The following member functions are declared in the protected interface:

```
ios();  
void init( streambuf * );  
void setstate( ios::iostate );
```

### Public Enumerations

The following enumeration typedefs are declared in the public interface:

```
typedef int iostate;  
typedef long fmtflags;  
typedef int openmode;  
typedef int seekdir;
```

### Public Member Functions

The following member functions are declared in the public interface:

```
ios( streambuf * );  
virtual ~ios();  
ostream *tie() const;  
ostream *tie( ostream * );  
streambuf *rdbuf() const;  
ios::iostate rdstate() const;  
ios::iostate clear( ios::iostate = 0 );  
int good() const;  
int bad() const;  
int fail() const;  
int eof() const;
```

```
ios::iostate exceptions( ios::iostate );
ios::iostate exceptions() const;
ios::fmtflags setf( ios::fmtflags, ios::fmtflags );
ios::fmtflags setf( ios::fmtflags );
ios::fmtflags unsetf( ios::fmtflags );
ios::fmtflags flags( ios::fmtflags );
ios::fmtflags flags() const;
char fill( char );
char fill() const;
int precision( int );
int precision() const;
int width( int );
int width() const;
long &iword( int );
void *&pword( int );
static void sync_with_stdio();
static ios::fmtflags bitalloc();
static int xalloc();
```

### **Public Member Operators**

The following member operators are declared in the public interface:

```
operator void *() const;
int operator !() const;
```

**See Also:** `iostream`, `istream`, `ostream`, `streambuf`

**Synopsis:**

```
#include <iostream.h>
public:
int ios::bad() const;
```

**Semantics:** The bad public member function queries the state of the ios object.

**Results:** The bad public member function returns a non-zero value if ios::badbit is set in the error state in the inherited ios object, otherwise zero is returned.

**See Also:** ios::clear, eof, fail, good, iostate, operator !, operator void \*, rdstate, setstate

## *ios::bitalloc()*

---

**Synopsis:**

```
#include <iostream.h>
public:
static ios::fmtflags ios::bitalloc();
```

**Semantics:** The `bitalloc` public static member function is used to allocate a new `ios::fmtflags` bit for use by user derived classes.

Because the `bitalloc` public static member function manipulates `static` member data, its behavior is not tied to any one object but affects the entire class of objects. The value that is returned by the `bitalloc` public static member function is valid for all objects of all classes derived from the `ios` class. No subsequent call to the `bitalloc` public static member function will return the same value as a previous call.

The bit value allocated may be used with the member functions that query and affect `ios::fmtflags`. In particular, the bit can be set with the `setf` or `flags` member functions or the `setiosflags` manipulator, and reset with the `unsetf` or `flags` member functions or the `resetiosflags` manipulator.

There are two constants defined in `<iostream.h>` which indicate the number of bits available when a program starts. `_LAST_FORMAT_FLAG` indicates the last bit used by the built-in format flags described by `ios::fmtflags`. `_LAST_FLAG_BIT` indicates the last bit that is available for the `bitalloc` public static member function to allocate. The difference between the bit positions indicates how many bits are available.

**Results:** The `bitalloc` public static member function returns the next available `ios::fmtflags` bit for use by user derived classes. If no more bits are available, zero is returned.

**See Also:** `ios::fmtflags`

**Synopsis:**

```
#include <iostream.h>
public:
    iostate ios::clear( ios::iostate flags = 0 );
```

**Semantics:** The `clear` public member function is used to change the current value of `ios::iostate` in the `ios` object. `ios::iostate` is cleared, all bits specified in *flags* are set.

**Results:** The `clear` public member function returns the previous value of `ios::iostate`.

**See Also:** `ios::bad`, `eof`, `fail`, `good`, `iostate`, `operator !`, `operator void *`, `rdstate`, `setstate`

## *ios::eof()*

---

**Synopsis:**

```
#include <iostream.h>
public:
int ios::eof() const;
```

**Semantics:** The `eof` public member function queries the state of the `ios` object.

**Results:** The `eof` public member function returns a non-zero value if `ios::eofbit` is set in the error state in the inherited `ios` object, otherwise zero is returned.

**See Also:** `ios::bad`, `clear`, `fail`, `good`, `iostate`, `rdstate`, `setstate`



**Synopsis:**

```
#include <iostream.h>
public:
ios::iostate ios::exceptions() const;
ios::iostate ios::exceptions( int enable );
```

**Semantics:** The `exceptions` public member function queries and/or sets the bits that control which exceptions are enabled. `ios::iostate` within the `ios` object is used to enable and disable exceptions.

When a condition arises that sets a bit in `ios::iostate`, a check is made to see if the same bit is also set in the exception bits. If so, an exception is thrown. Otherwise, no exception is thrown.

The first form of the `exceptions` public member function looks up the current setting of the exception bits. The bit values are those described by `ios::iostate`.

The second form of the `exceptions` public member function sets the exceptions bits to those specified in the *enable* parameter, and returns the current settings.

**Results:** The `exceptions` public member function returns the previous setting of the exception bits.

**See Also:** `ios::clear`, `iostate`, `rdstate`, `setstate`

## *ios::fail()*

---

**Synopsis:**

```
#include <iostream.h>
public:
int ios::fail() const;
```

**Semantics:** The `fail` public member function queries the state of the `ios` object.

**Results:** The `fail` public member function returns a non-zero value if `ios::failbit` or `ios::badbit` is set in the error state in the inherited `ios` object, otherwise zero is returned.

**See Also:** `ios::bad`, `clear`, `eof`, `good`, `iostate`, `operator !`, `operator void *`, `rdstate`, `setstate`

**Synopsis:**

```
#include <iostream.h>
public:
char ios::fill() const;
char ios::fill( char fillchar );
```

**Semantics:** The `fill` public member function queries and/or sets the *fill character* used when the size of a formatted object is smaller than the *format width* specified.

The first form of the `fill` public member function looks up the current value of the *fill character*.

The second form of the `fill` public member function sets the *fill character* to *fillchar*.

By default, the *fill character* is a space.

**Results:** The `fill` public member function returns the previous value of the *fill character*.

**See Also:** `ios::fmtflags`, manipulator `setfill`

## *ios::flags()*

---

**Synopsis:**

```
#include <iostream.h>
public:
ios::fmtflags ios::flags() const;
ios::fmtflags ios::flags( ios::fmtflags setbits );
```

**Semantics:** The `flags` public member function is used to query and/or set the value of `ios::fmtflags` in the `ios` object.

The first form of the `flags` public member function looks up the current `ios::fmtflags` value.

The second form of the `flags` public member function sets `ios::fmtflags` to the value specified in the *setbits* parameter.

Note that the `setf` public member function only turns bits on, while the `flags` public member function turns some bits on and some bits off.

**Results:** The `flags` public member function returns the previous `ios::fmtflags` value.

**See Also:** `ios::fmtflags`, `setf`, `unsetf`, `manipulator dec`, `manipulator hex`, `manipulator oct`, `manipulator resetiosflags`, `manipulator setbase`, `manipulator setiosflags`

**Synopsis:**

```
#include <iostream.h>
public:
enum fmt_flags {
skipws = 0x0001, // skip whitespace
left = 0x0002, // align field to left edge
right = 0x0004, // align field to right edge
internal = 0x0008, // sign at left, value at right
dec = 0x0010, // decimal conversion for integers
oct = 0x0020, // octal conversion for integers
hex = 0x0040, // hexadecimal conversion for integers
showbase = 0x0080, // show dec/octal/hex base on output
showpoint = 0x0100, // show decimal and digits on output
uppercase = 0x0200, // use uppercase for format characters
showpos = 0x0400, // use + for output positive numbers
scientific = 0x0800, // use scientific notation for output
fixed = 0x1000, // use floating notation for output
unitbuf = 0x2000, // flush stream after output
stdio = 0x4000, // flush stdout/stderr after output

basefield = dec | oct | hex,
adjustfield= left | right | internal,
floatfield = scientific | fixed
};
typedef long fmtflags;
```

**Semantics:** The type `ios::fmt_flags` is a set of bits representing methods of formatting objects written to the stream and interpreting objects read from the stream. The `ios::fmtflags` member typedef represents the same set of bits, but uses a `long` to represent the values, thereby avoiding problems made possible by the compiler's ability to use smaller types for enumerations. All uses of these bits should use the `ios::fmtflags` member typedef.

The bit values defined by the `ios::fmtflags` member typedef are set and read by the member functions `setf`, `unsetf` and `flags`, as well as the manipulators `setiosflags` and `resetiosflags`.

Because one field is used to store all of these bits, there are three special values used to mask various groups of bits. These values are named `ios::basefield`, `ios::adjustfield` and `ios::floatfield`, and are discussed with the bits that they are used to mask.

`ios::skipws` controls whether or not whitespace characters are automatically skipped when using an operator `>>` extractor. If `ios::skipws` is on, any use of the operator `>>` extractor skips whitespace characters before inputting the next item. Otherwise, skipping of whitespace characters must be handled by the program.

`ios::left`, `ios::right` and `ios::internal` control the alignment of items written using an operator `<<` inserter. These bits are usually used in conjunction with the *format width* and *fill character*.

`ios::adjustfield` can be used to mask the alignment bits returned by the `setf`, `unsetf` and `flags` member functions, and for setting new values to ensure that no other bits are accidentally affected.

When the item to be written is smaller than the *format width* specified, *fill characters* are written to occupy the additional space. If `ios::left` is in effect, the item is written in the left portion of the available space, and *fill characters* are written in the right portion. If `ios::right` is in effect, the item is written in the right portion of the available space, and *fill characters* are written in the left portion. If `ios::internal` is in effect, any sign character or base indicator is written in the left portion, the digits are written in the right portion, and *fill characters* are written in between.

If no alignment is specified, `ios::right` is assumed.

If the item to be written is as big as or bigger than the *format width* specified, no *fill characters* are written and the alignment is ignored.

`ios::dec`, `ios::oct` and `ios::hex` control the base used to format integers being written to the stream, and also control the interpretation of integers being read from the stream.

`ios::basefield` can be used to mask the base bits returned by the member functions `setf`, `unsetf` and `flags`, and for setting new values to ensure that no other bits are accidentally affected.

When an integer is being read from the stream, these bits control the base used for the interpretation of the digits. If none of these bits is set, a number that starts with `0x` or `0X` is interpreted as hexadecimal (digits `0123456789`, plus the letters `abcdef` or `ABCDEF`), a number that starts with `0` (zero) is interpreted as octal (digits `01234567`), otherwise the number is interpreted as decimal (digits `0123456789`). If one of the bits is set, then the prefix is not necessary and the number is interpreted according to the bit.

When any one of the integer types is being written to the stream, it can be written in decimal, octal or hexadecimal. If none of these bits is set, `ios::dec` is assumed.

If `ios::dec` is set (or assumed), the integer is written in decimal (digits `0123456789`). No prefix is included.

If `ios::oct` is set, the integer is written in octal (digits `01234567`). No sign character is written, as the number is treated as an unsigned quantity upon conversion to octal.

If `ios::hex` is set, the integer is written in hexadecimal (digits 0123456789, plus the letters abcdef or ABCDEF, depending on the setting of `ios::uppercase`). No sign character is written, as the number is treated as an unsigned quantity upon conversion to hexadecimal.

`ios::showbase` controls whether or not integers written to the stream in octal or hexadecimal form have a prefix that indicates the base of the number. If the bit is set, decimal numbers are written without a prefix, octal numbers are written with the prefix 0 (zero) and hexadecimal numbers are written with the prefix 0x or 0X depending on the setting of `ios::uppercase`. If the `ios::showbase` is not set, no prefixes are written.

`ios::showpoint` is used to control whether or not the decimal point and trailing zeroes are trimmed when floating-point numbers are written to the stream. If the bit is set, no trimming is done, causing the number to appear with the specified *format precision*. If the bit is not set, any trailing zeroes after the decimal point are trimmed, and if not followed by any digits, the decimal point is removed as well.

`ios::uppercase` is used to force to upper-case all letters used in formatting numbers, including the letter-digits abcdef, the x hexadecimal prefix, and the e used for the exponents in floating-point numbers.

`ios::showpos` controls whether or not a + is added to the front of positive integers being written to the stream. If the bit is set, the number is positive and the number is being written in decimal, a + is written before the first digit.

`ios::scientific` and `ios::fixed` controls the form used for writing floating-point numbers to the stream. Floating-point numbers can be written in scientific notation (also called exponential notation) or in fixed-point notation.

`ios::floatfield` can be used to mask the floating-format bits returned by the member functions `setf`, `unsetf` and `flags`, and for setting new values to ensure that no other bits are accidentally affected.

If `ios::scientific` is set, the floating-point number is written with a leading - sign (for negative numbers), a digit, a decimal point, more digits, an e (or E if `ios::uppercase` is set), a + or - sign, and two or three digits representing the exponent. The digit before the decimal is not zero unless the number is zero. The total number of digits before and after the decimal is equal to the specified *format precision*. If `ios::showpoint` is not set, trimming of the decimal and digits following the decimal may occur.

If `ios::fixed` is set, the floating-point number is written with a - sign (for negative numbers), at least one digit, the decimal point, and as many digits following the decimal as specified by the *format precision*. If `ios::showpoint` is not set, trimming of the decimal and digits following the decimal may occur.

If neither `ios::scientific` nor `ios::fixed` is specified, the floating-point number is formatted using scientific notation provided one or both of the following conditions are met:

- the exponent is less than -4, or,
- the exponent is greater than the *format precision*.

Otherwise, fixed-point notation is used.

`ios::unitbuf` controls whether or not the stream is flushed after each item is written. If the bit is set, every item that is written to the stream is followed by a flush operation, which ensures that the I/O stream buffer associated with the stream is kept empty, immediately transferring the data to its final destination.

`ios::stdio` controls whether or not the stream is synchronized after each item is written. If the bit is set, every item that is written to the stream causes the stream to be synchronized, which means any input or output buffers are flushed so that an I/O operation performed using C (not C++) I/O behaves in an understandable way. If the output buffer was not flushed, writing using C++ and then C I/O functions could cause the output from the C functions to appear before the output from the C++ functions, since the characters might be sitting in the C++ output buffer. Similarly, after the C output operations are done, a call should be made to the C library `fflush` function on the appropriate stream before resuming C++ output operations.

**See Also:** `ios::flags`, `setf`, `unsetf`, `manipulator dec`, `manipulator hex`, `manipulator oct`, `manipulator resetiosflags`, `manipulator setbase`, `manipulator setiosflags`



**Synopsis:**

```
#include <iostream.h>
public:
int ios::good() const;
```

**Semantics:** The `good` public member function queries the state of the `ios` object.

**Results:** The `good` public member function returns a non-zero value if none of `ios::iostate` is clear, otherwise zero is returned.

**See Also:** `ios::bad`, `clear`, `eof`, `fail`, `iostate`, `rdstate`, `setstate`

## *ios::init()*

---

**Synopsis:**

```
#include <iostream.h>
protected:
void ios::init( streambuf *sb );
```

**Semantics:** The `init` public protected member function is used by derived classes to explicitly initialize the `ios` portion of the derived object, and to associate a `streambuf` with the `ios` object. The `init` public protected member function performs the following steps:

1. The default *fill character* is set to a space.
2. The *format precision* is set to six.
3. The `streambuf` pointer (returned by the `rdbuf` member function) is set to *sb*.
4. The remaining fields of the `ios` object are initialized to zero.

**Results:** If *sb* is `NULL` the `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `ios`, `rdbuf`

**Synopsis:** `#include <iostream.h>`  
`protected:`  
`ios::ios();`

**Semantics:** This form of the protected `ios` constructor creates a default `ios` object that is initialized, but does not have an associated `streambuf`. Initialization of an `ios` object is handled by the `init` protected member function.

**Results:** This protected `ios` constructor creates an `ios` object and sets `ios::badbit` in the error state in the inherited `ios` object.

**See Also:** `~ios`, `init`

## *ios::ios()*

---

**Synopsis:**

```
#include <iostream.h>
public:
ios::ios( streambuf *sb );
```

**Semantics:** This form of the public `ios` constructor creates an `ios` object that is initialized and has an associated `streambuf`. Initialization of an `ios` object is handled by the `init` protected member function. Once the `init` protected member function is completed, the `ios` object's `streambuf` pointer is set to `sb`. If `sb` is not `NULL`, `ios::badbit` is cleared from the error state in the inherited `ios` object.

**Results:** This public `ios` constructor creates an `ios` object and, if `sb` is `NULL`, sets `ios::badbit` in the error state in the inherited `ios` object.

**See Also:** `~ios`, `init`

**Synopsis:** `#include <iostream.h>`  
`public:`  
`virtual ios::~~ios();`

**Semantics:** The public virtual `~ios` destructor destroys an `ios` object. The call to the public virtual `~ios` destructor is inserted implicitly by the compiler at the point where the `ios` object goes out of scope.

**Results:** The `ios` object is destroyed.

**See Also:** `ios`

**Synopsis:**

```
#include <iostream.h>
public:
enum io_state {
goodbit = 0x00, // no errors
badbit = 0x01, // operation failed, may not proceed
failbit = 0x02, // operation failed, may proceed
eofbit = 0x04 // end of file encountered
};
typedef int iostate;
```

**Semantics:** The type `ios::io_state` is a set of bits representing the current state of the stream. The `ios::iostate` member typedef represents the same set of bits, but uses an `int` to represent the values, thereby avoiding problems made possible by the compiler's ability to use smaller types for enumerations. All uses of these bits should use the `ios::iostate` member typedef.

The bit values defined by the `ios::iostate` member typedef can be read and set by the member functions `rdstate` and `clear`, and can be used to control exception handling with the member function exceptions.

`ios::badbit` represents the state where the stream is no longer usable because of some error condition.

`ios::failbit` represents the state where the previous operation on the stream failed, but the stream is still usable. Subsequent operations on the stream are possible, but the state must be cleared using the `clear` member function.

`ios::eofbit` represents the state where the end-of-file condition has been encountered. The stream may still be used, but the state must be cleared using the `clear` member function.

Even though `ios::goodbit` is not a bit value (because its value is zero, which has no bits on), it is provided for completeness.

**See Also:** `ios::bad`, `clear`, `eof`, `fail`, `good`, `operator !`, `operator void *`, `rdstate`, `setstate`

**Synopsis:**

```
#include <iostream.h>
public:
long &ios::iword( int index );
```

**Semantics:** The `iword` public member function creates a reference to a `long int`, which may be used to store and retrieve any suitable integer value. The *index* parameter specifies which `long int` is to be referenced and must be obtained from a call to the `xalloc` static member function.

Note that the `iword` and `pword` public member functions return references to the same storage with a different type. Therefore, each *index* obtained from the `xalloc` static member function can be used only for an integer or a pointer, not both.

Since the `iword` public member function returns a reference and the `ios` class cannot predict how many such items will be required by a program, it should be assumed that each call to the `xalloc` static member function invalidates all previous references returned by the `iword` public member function. Therefore, the `iword` public member function should be called each time the reference is needed.

**Results:** The `iword` public member function returns a reference to a `long int`.

**See Also:** `ios::pword`, `xalloc`

**Synopsis:**

```
#include <iostream.h>
public:
enum open_mode {
in = 0x0001, // open for input
out = 0x0002, // open for output
atend = 0x0004, // seek to end after opening
append = 0x0008, // open for output, append to the end
truncate = 0x0010, // discard contents after opening
noreplace = 0x0020, // open only an existing file
noreplace = 0x0040, // open only a new file
text = 0x0080, // open as text file
binary = 0x0100, // open as binary file

app = append, // synonym
ate = atend, // synonym
trunc = truncate // synonym
};
typedef int openmode;
```

**Semantics:** The type `ios::open_mode` is a set of bits representing ways of opening a stream. The `ios::openmode` member typedef represents the same set of bits, but uses an `int` to represent the values, thereby avoiding problems made possible by the compiler's ability to use smaller types for enumerations. All uses of these bits should use the `ios::openmode` member typedef.

The bit values defined by `ios::openmode` member typedef can be specified in the constructors for stream objects, as well as in various member functions.

`ios::in` is specified in a stream for which input operations may be performed. `ios::out` is specified in a stream for which output operations may be performed. A stream for which only `ios::in` is specified is referred to as an *input* stream. A stream for which only `ios::out` is specified is referred to as an *output* stream. A stream where both `ios::in` and `ios::out` are specified is referred to as an *input/output* stream.

`ios::atend` and `ios::ate` are equivalent, and either one is specified for streams that are to be positioned to the end before the first operation takes place. `ios::ate` is provided for historical purposes and compatibility with other implementations of I/O streams. Note that this bit positions the stream to the end exactly once, when the stream is opened.

`ios::append` and `ios::app` are equivalent, and either one is specified for streams that are to be positioned to the end before any and all output operations take place. `ios::app` is provided for historical purposes and compatibility with other implementations of I/O streams. Note that this bit causes the stream to be positioned to the end before each output operation, while `ios::atend` causes the stream to be positioned to the end only when first opened.



`ios::truncate` and `ios::trunc` are equivalent, and either one is specified for streams that are to be truncated to zero length before the first operation takes place. `ios::trunc` is provided for historical purposes and compatibility with other implementations of I/O streams.

`ios::nocreate` is specified if the file must exist before it is opened. If the file does not exist, an error occurs.

`ios::noreplace` is specified if the file must not exist before it is opened. That is, the file must be a new file. If the file exists, an error occurs.

`ios::text` is specified if the file is to be treated as a *text* file. A text file is divided into records, and each record is terminated by a *new-line* character, usually represented as `'\n'`. The new-line character is translated into a form that is compatible with the underlying file system's concept of text files. This conversion happens automatically whenever the new-line is written to the file, and the inverse conversion (to the new-line character) happens automatically whenever the end of a record is read from the file system.

`ios::binary` is specified if the file is to be treated as a *binary* file. Binary files are streams of characters. No character has a special meaning. No grouping of characters into records is apparent to the program, although the underlying file system may cause such a grouping to occur.

The following default behaviors are defined:

If `ios::out` is specified and none of `ios::in`, `ios::append` or `ios::atend` are specified, `ios::truncate` is assumed.

If `ios::append` is specified, `ios::out` is assumed.

If `ios::truncate` is specified, `ios::out` is assumed.

If neither `ios::text` nor `ios::binary` is specified, `ios::text` is assumed.

## *ios::operator !()*

---

**Synopsis:**

```
#include <iostream.h>
public:
int ios::operator !() const;
```

**Semantics:** The `operator !` public member function tests the error state in the inherited `ios` object of the `ios` object.

**Results:** The `operator !` public member function returns a non-zero value if either of `ios::failbit` or `ios::badbit` bits are set in the error state in the inherited `ios` object, otherwise zero is returned.

**See Also:** `ios::bad`, `clear`, `fail`, `good`, `iostate`, `operator void *`, `rdstate`, `setstate`

**Synopsis:** `#include <iostream.h>`  
`public:`  
`ios::operator void *() const;`

**Semantics:** The `operator void *` public member function converts the `ios` object into a pointer to `void`. The actual pointer value returned is meaningless and intended only for comparison with `NULL` to determine the error state in the inherited `ios` object of the `ios` object.

**Results:** The `operator void *` public member function returns a `NULL` pointer if either of `ios::failbit` or `ios::badbit` bits are set in the error state in the inherited `ios` object, otherwise a non- `NULL` pointer is returned.

**See Also:** `ios::bad`, `clear`, `fail`, `good`, `iostate`, `operator !`, `rdstate`, `setstate`

## *ios::precision()*

---

**Synopsis:**

```
#include <iostream.h>
public:
int ios::precision() const;
int ios::precision( int prec );
```

**Semantics:** The `precision` public member function is used to query and/or set the *format precision*. The *format precision* is used to control the number of digits of precision used when formatting floating-point numbers. For scientific notation, the *format precision* describes the total number of digits before and after the decimal point, but not including the exponent. For fixed-point notation, the *format precision* describes the number of digits after the decimal point.

The first form of the `precision` public member function looks up the current *format precision*.

The second form of the `precision` public member function sets the *format precision* to *prec*.

By default, the *format precision* is six. If *prec* is specified to be less than zero, the *format precision* is set to six. Otherwise, the specified *format precision* is used. For scientific notation, a *format precision* of zero is treated as a precision of one.

**Results:** The `precision` public member function returns the previous *format precision* setting.

**See Also:** `ios::fmtflags`, manipulator `setprec`

**Synopsis:**

```
#include <iostream.h>
public:
void * &ios::pword( int index );
```

**Semantics:** The `pword` public member function creates a reference to a `void` pointer, which may be used to store and retrieve any suitable pointer value. The *index* parameter specifies which `void` pointer is to be referenced and must be obtained from a call to the `xalloc` static member function.

Note that the `iword` and `pword` public member functions return references to the same storage with a different type. Therefore, each *index* obtained from the `xalloc` static member function can be used only for an integer or a pointer, not both.

Since the `pword` public member function returns a reference and the `ios` class cannot predict how many such items will be required by a program, it should be assumed that each call to the `xalloc` static member function invalidates all previous references returned by the `pword` public member function. Therefore, the `pword` public member function should be called each time the reference is needed.

**Results:** The `pword` public member function returns a reference to a `void` pointer.

**See Also:** `ios::iword`, `xalloc`

## *ios::rdbuf()*

---

**Synopsis:**

```
#include <iostream.h>
public:
    streambuf *ios::rdbuf() const;
```

**Semantics:** The `rdbuf` public member function looks up the pointer to the `streambuf` object which maintains the buffer associated with the `ios` object.

**Results:** The `rdbuf` public member function returns the pointer to the `streambuf` object associated with the `ios` object. If there is no associated `streambuf` object, `NULL` is returned.

**Synopsis:** `#include <iostream.h>`  
`public:`  
`iosstate ios::rdstate() const;`

**Semantics:** The `rdstate` public member function is used to query the current value of `ios::iosstate` in the `ios` object without modifying it.

**Results:** The `rdstate` public member function returns the current value of `ios::iosstate`.

**See Also:** `ios::bad`, `clear`, `eof`, `fail`, `good`, `iosstate`, `operator !`, `operator void *`, `setstate`

**Synopsis:**

```
#include <iostream.h>
public:
enum seek_dir {
beg, // seek from beginning
cur, // seek from current position
end // seek from end
};
typedef int seekdir;
```

**Semantics:** The type `ios::seek_dir` is a set of bits representing different methods of seeking within a stream. The `ios::seekdir` member typedef represents the same set of bits, but uses an `int` to represent the values, thereby avoiding problems made possible by the compiler's ability to use smaller types for enumerations. All uses of these bits should use the `ios::seekdir` member typedef.

The bit values defined by `ios::seekdir` member typedef are used by the member functions `seekg` and `seekp`, as well the `seekoff` and `seekpos` member functions in classes derived from the `streambuf` class.

`ios::beg` causes the seek offset to be interpreted as an offset from the beginning of the stream. The offset is specified as a positive value.

`ios::cur` causes the seek offset to be interpreted as an offset from the current position of the stream. If the offset is a negative value, the seek is towards the start of the stream. Otherwise, the seek is towards the end of the stream.

`ios::end` causes the seek offset to be interpreted as an offset from the end of the stream. The offset is specified as a negative value.



**Synopsis:**

```
#include <iostream.h>
public:
ios::fmtflags ios::setf( ios::fmtflags onbits );
ios::fmtflags ios::setf( ios::fmtflags setbits,
ios::fmtflags mask );
```

**Semantics:** The `setf` public member function is used to set bits in `ios::fmtflags` in the `ios` object.

The first form is used to turn on the bits that are on in the *onbits* parameter. (*onbits* is or'ed into `ios::fmtflags`).

The second form is used to turn off the bits specified in the *mask* parameter and turn on the bits specified in the *setbits* parameter. This form is particularly useful for setting the bits described by the `ios::basefield`, `ios::adjustfield` and `ios::floatfield` values, where only one bit should be on at a time.

**Results:** Both forms of the `setf` public member function return the previous `ios::fmtflags` value.

**See Also:** `ios::fmtflags`, `setf`, `unsetf`, `manipulator dec`, `manipulator hex`, `manipulator oct`, `manipulator setbase`, `manipulator setiosflags`, `manipulator resetiosflags`

## ***ios::setstate()***

---

**Synopsis:** `#include <iostream.h>`  
`protected:`  
`void ios::setstate( int or_bits );`

**Semantics:** The `setstate` protected member function is provided as a convenience for classes derived from the `ios` class. It turns on the error state in the inherited `ios` object bits that are set in the `or_bits` parameter, and leaves the other error state in the inherited `ios` object bits unchanged.

**Results:** The `setstate` protected member function sets the bits specified by `or_bits` in the error state in the inherited `ios` object.

**See Also:** `ios::bad`, `clear`, `eof`, `fail`, `good`, `iostate`, `operator !`, `operator void *`, `rdstate`

**Synopsis:**

```
#include <iostream.h>
public:
static void ios::sync_with_stdio();
```

**Semantics:** The `sync_with_stdio` public static member function is obsolete. It is provided for compatibility.

**Results:** The `sync_with_stdio` public static member function has no return value.

## *ios::tie()*

---

**Synopsis:**

```
#include <iostream.h>
public:
ostream *ios::tie() const;
ostream *ios::tie( ostream *ostrm );
```

**Semantics:** The `tie` public member function is used to query and/or set up a connection between the `ios` object and another stream. The connection causes the output stream specified by `ostrm` to be flushed whenever the `ios` object is about to read characters from a device or is about to write characters to an output buffer or device.

The first form of the `tie` public member function is used to query the current tie.

The second form of the `tie` public member function is used to set the tied stream to `ostrm`.

Normally, the predefined streams `cin` and `cerr` set up ties to `cout` so that any input from the terminal flushes any buffered output, and any writes to `cerr` flush `cout` before the characters are written. `cout` does not set up a tie to `cerr` because `cerr` has the flag `ios::unitbuf` set, so it flushes itself after every write operation.

**Results:** Both forms of the `tie` public member function return the previous tie value.

**See Also:** `ios::fmtflags`

**Synopsis:**

```
#include <iostream.h>
public:
ios::fmtflags ios::unsetf( ios::fmtflags offbits );
```

**Semantics:** The `unsetf` public member function is used to turn off bits in `ios::fmtflags` that are set in the *offbits* parameter. All other bits in `ios::fmtflags` are unchanged.

**Results:** The `unsetf` public member function returns the old `ios::fmtflags` value.

**See Also:** `ios::fmtflags`, `setf`, `unsetf`, manipulator `dec`, manipulator `hex`, manipulator `oct`, manipulator `setbase`, manipulator `setiosflags`, manipulator `resetiosflags`

## *ios::width()*

---

**Synopsis:**

```
#include <iostream.h>
public:
int ios::width() const;
int ios::width( int wid );
```

**Semantics:** The `width` public member function is used to query and/or set the *format width* used to format the next item. A *format width* of zero indicates that the item is to be written using exactly the number of positions required. Other values indicate that the item must occupy at least that many positions. If the formatted item is larger than the specified *format width*, the *format width* is ignored and the item is formatted using the required number of positions.

The first form of the `width` public member function is used to query the *format width* that is to be used for the next item.

The second form of the `width` public member function is used to set the *format width* to *wid* for the next item to be formatted.

After an item has been formatted, the *format width* is reset to zero. Therefore, any non-zero *format width* must be set before each item that is to be formatted.

**Results:** The `width` public member function returns the previous *format width*.

**See Also:** `ios::fmtflags`, manipulator `setw`, manipulator `setwidth`

**Synopsis:**

```
#include <iostream.h>
public:
static int ios::xalloc();
```

**Semantics:** The `xalloc` public static member function returns an index into an array of items that the program may use for any purpose. Each item can be either a `long int` or a pointer to `void`. The index can be used with the `iword` and `pword` member functions.

Because the `xalloc` public static member function manipulates `static` member data, its behavior is not tied to any one object but affects the entire class of objects. The value that is returned by the `xalloc` public static member function is valid for all objects of all classes derived from the `ios` class. No subsequent call to the `xalloc` public static member function will return the same value as a previous call.

**Results:** The `xalloc` public static member function returns an index for use with the `iword` and `pword` member functions.

**See Also:** `ios::iword`, `pword`

## ***iostream***

---

**Declared:** `iostream.h`

**Derived from:**  
`istream, ostream`

**Derived by:** `fstream, stringstream`

The `iostream` class supports reading and writing of characters from and to the standard input/output devices, usually the keyboard and screen. The `iostream` class provides formatted conversion of characters to and from other types (e.g. integers and floating-point numbers). The associated `streambuf` class provides the methods for communicating with the actual device, while the `istream` class provides the interpretation of the characters.

Generally, an `iostream` object won't be created by a program, since there is no mechanism at this level to "open" a device. No instance of an `iostream` object is created by default, since it is usually not possible to perform both input and output on the standard input/output devices. The `iostream` class is provided as a base class for other derived classes that can provide both input and output capabilities through the same object. The `fstream` and `stringstream` classes are examples of classes derived from the `iostream` class.

### **Protected Member Functions**

The following protected member functions are declared:

```
iostream();
```

### **Public Member Functions**

The following public member functions are declared:

```
iostream( ios const & );  
iostream( streambuf * );  
virtual ~iostream();
```

### **Public Member Operators**

The following public member operators are declared:

```
iostream & operator =( streambuf * );  
iostream & operator =( ios const & );
```

**See Also:** `ios, istream, ostream`

## **710 *Input/Output Classes***



**Synopsis:** `#include <iostream.h>`  
`protected:`  
`iostream::iostream();`

**Semantics:** This form of the protected `iostream` constructor creates an `iostream` object without an attached `streambuf` object.

This form of the protected `iostream` constructor is only used implicitly by the compiler when it generates a constructor for a derived class.

**Results:** The protected `iostream` constructor produces an initialized `iostream` object. `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~iostream`

## *iostream::iostream()*

---

**Synopsis:**

```
#include <iostream.h>
public:
    iostream::iostream( ios const &strm );
```

**Semantics:** This form of the public `iostream` constructor creates an `iostream` object associated with the `streambuf` object currently associated with the `strm` parameter. The `iostream` object is initialized and will use the `strm` `streambuf` object for subsequent operations. `strm` will continue to use the `streambuf` object.

**Results:** The public `iostream` constructor produces an initialized `iostream` object. If there is no `streambuf` object currently associated with the `strm` parameter, `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~iostream`

**Synopsis:**

```
#include <iostream.h>
public:
    iostream::iostream( ostreambuf *sb );
```

**Semantics:** This form of the public `iostream` constructor creates an `iostream` object with an attached `ostreambuf` object.

Since a user program usually will not create an `iostream` object, this form of the public `iostream` constructor is unlikely to be explicitly used, except in the member initializer list for the constructor of a derived class. The `sb` parameter is a pointer to a `ostreambuf` object, which should be connected to the source and sink of characters for the stream.

**Results:** The public `iostream` constructor produces an initialized `iostream` object. If the `sb` parameter is `NULL`, `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~iostream`

## *iostream::~~iostream()*

---

**Synopsis:**

```
#include <iostream.h>
public:
virtual iostream::~~iostream();
```

**Semantics:** The public `~iostream` destructor does not do anything explicit. The `ios` destructor is called for that portion of the `iostream` object. The call to the public `~iostream` destructor is inserted implicitly by the compiler at the point where the `iostream` object goes out of scope.

**Results:** The `iostream` object is destroyed.

**See Also:** `iostream`

**Synopsis:**

```
#include <iostream.h>
public:
iostream &iostream::operator =( streambuf *sb );
```

**Semantics:** This form of the `operator =` public member function initializes the target `iostream` object and sets up an association between the `iostream` object and the `streambuf` object specified by the `sb` parameter.

**Results:** The `operator =` public member function returns a reference to the `iostream` object that is the target of the assignment. If the `sb` parameter is `NULL`, `ios::badbit` is set in the error state in the inherited `ios` object.

## *iostream::operator =()*

---

**Synopsis:**

```
#include <iostream.h>
public:
    ostream &iostream::operator =( const ios &strm );
```

**Semantics:** This form of the `operator =` public member function initializes the `ostream` object and sets up an association between the `ostream` object and the `streambuf` object currently associated with the `strm` parameter.

**Results:** The `operator =` public member function returns a reference to the `ostream` object that is the target of the assignment. If there is no `streambuf` object currently associated with the `strm` parameter, `ios::badbit` is set in the error state in the inherited `ios` object.

**Declared:** iostream.h

**Derived from:**  
ios

**Derived by:** istream, ifstream, istrstream

The `istream` class supports reading characters from a class derived from `streambuf`, and provides formatted conversion of characters into other types (such as integers and floating-point numbers). The `streambuf` class provides the methods for communicating with the external device (keyboard, disk), while the `istream` class provides the interpretation of the resulting characters.

Generally, an `istream` object won't be explicitly created by a program, since there is no mechanism at this level to open a device. The only default `istream` object in a program is `cin`, which reads from standard input (usually the keyboard).

The `istream` class supports two basic concepts of input: formatted and unformatted. The overloaded operator `>>` member functions are called *extractors* and they provide the support for formatted input. The rest of the member functions deal with unformatted input, managing the state of the `ios` object and providing a friendlier interface to the associated `streambuf` object.

### Protected Member Functions

The following protected member functions are declared:

```
istream();  
eatwhite();
```

### Public Member Functions

The following public member functions are declared:

```
istream( istream const & );  
istream( streambuf * );  
virtual ~istream();  
int ipfx( int = 0 );  
void isfx();  
int get();  
istream &get( char *, int, char = '\n' );  
istream &get( signed char *, int, char = '\n' );  
istream &get( unsigned char *, int, char = '\n' );  
istream &get( char & );  
istream &get( signed char & );
```

```
istream &get( unsigned char & );
istream &get( streambuf &, char = '\n' );
istream &getline( char *, int, char = '\n' );
istream &getline( signed char *, int, char = '\n' );
istream &getline( unsigned char *, int, char = '\n' );
istream &ignore( int = 1, int = EOF );
istream &read( char *, int );
istream &read( signed char *, int );
istream &read( unsigned char *, int );
istream &seekg( streampos );
istream &seekg( streamoff, ios::seekdir );
istream &putback( char );
streampos tellg();
int gcount() const;
int peek();
int sync();
```

### **Public Member Operators**

The following public member operators are declared:

```
istream &operator =( streambuf * );
istream &operator =( istream const & );
istream &operator >>( char * );
istream &operator >>( signed char * );
istream &operator >>( unsigned char * );
istream &operator >>( char & );
istream &operator >>( signed char & );
istream &operator >>( unsigned char & );
istream &operator >>( signed short & );
istream &operator >>( unsigned short & );
istream &operator >>( signed int & );
istream &operator >>( unsigned int & );
istream &operator >>( signed long & );
istream &operator >>( unsigned long & );
istream &operator >>( float & );
istream &operator >>( double & );
istream &operator >>( long double & );
istream &operator >>( streambuf & );
istream &operator >>( istream &(*) ( istream & ) );
istream &operator >>( ios &(*) ( ios & ) );
```

**See Also:** `ios`, `iostream`, `ostream`



**Synopsis:** `#include <iostream.h>`  
`protected:`  
`void istream::eatwhite();`

**Semantics:** The `eatwhite` protected member function extracts and discards whitespace characters from the `istream` object, until a non-whitespace character is found. The non-whitespace character is not extracted.

**Results:** The `eatwhite` protected member function sets `ios::eofbit` in the error state in the inherited `ios` object if end-of-file is encountered as the first character while extracting whitespace characters.

**See Also:** `istream::ignore`, `ios::fmtflags`

## *istream::gcount()*

---

**Synopsis:**

```
#include <iostream.h>
public:
int istream::gcount() const;
```

**Semantics:** The `gcount` public member function determines the number of characters extracted by the last unformatted input member function.

**Results:** The `gcount` public member function returns the number of characters extracted by the last unformatted input member function.

**See Also:** `istream::get`, `getline`, `read`

**Synopsis:**

```
#include <iostream.h>
public:
int istream::get();
```

**Semantics:** This form of the `get` public member function performs an unformatted read of a single character from the `istream` object.

**Results:** This form of the `get` public member function returns the character read from the `istream` object. If the `istream` object is positioned at end-of-file before the read, `EOF` is returned and `ios::eofbit` bit is set in the error state in the inherited `ios` object. `ios::failbit` bit is not set by this form of the `get` public member function.

**See Also:** `istream::putback`

## *istream::get()*

---

**Synopsis:**

```
#include <iostream.h>
public:
    istream &istream::get( char &ch );
    istream &istream::get( signed char &ch );
    istream &istream::get( unsigned char &ch );
```

**Semantics:** These forms of the `get` public member function perform an unformatted read of a single character from the `istream` object and store the character in the `ch` parameter.

**Results:** These forms of the `get` public member function return a reference to the `istream` object. `ios::eofbit` is set in the error state in the inherited `ios` object if the `istream` object is positioned at end-of-file before the attempt to read the character. `ios::failbit` is set in the error state in the inherited `ios` object if no character is read.

**See Also:** `istream::read`, operator `>>`

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::get( char *buf, int len,
char delim = '\n' );
istream &istream::get( signed char *buf, int len,
char delim = '\n' );
istream &istream::get( unsigned char *buf, int len,
char delim = '\n' );
```

**Semantics:** These forms of the `get` public member function perform an unformatted read of at most *len* -1 characters from the `istream` object and store them starting at the memory location specified by the *buf* parameter. If the character specified by the *delim* parameter is encountered in the `istream` object before *len* -1 characters have been read, the read terminates without extracting the delimiting character.

After the read terminates, whether or not an error occurred, a null character is stored in *buf* following the last character read from the `istream` object.

If the *delim* parameter is not specified, the new-line character is assumed.

**Results:** These forms of the `get` public member function return a reference to the `istream` object. If end-of-file is encountered as the first character, `ios::eofbit` is set in the error state in the inherited `ios` object. If no characters are stored into *buf*, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `istream::getline`, `read`, `operator >>`

## *istream::get()*

---

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::get( streambuf &sb, char delim = '\n' );
```

**Semantics:** This form of the `get` public member function performs an unformatted read of characters from the `istream` object and transfers them to the `streambuf` object specified in the `sb` parameter. The transfer stops if end-of-file is encountered, the delimiting character specified in the `delim` parameter is found, or if the store into the `sb` parameter fails. If the `delim` character is found, it is not extracted from the `istream` object and is not transferred to the `sb` object.

If the `delim` parameter is not specified, the new-line character is assumed.

**Results:** The `get` public member function returns a reference to the `istream` object. `ios::failbit` is set in the error state in the inherited `ios` object if the store into the `streambuf` object fails.

**See Also:** `istream::getline`, `read`, `operator >>`

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::getline( char *buf, int len,
char delim = '\n' );
istream &istream::getline( signed char *buf, int len,
char delim = '\n' );
istream &istream::getline( unsigned char *buf, int len,
char delim = '\n' );
```

**Semantics:** The `getline` public member function performs an unformatted read of at most `len - 1` characters from the `istream` object and stores them starting at the memory location specified by the `buf` parameter. If the delimiting character, specified by the `delim` parameter, is encountered in the `istream` object before `len - 1` characters have been read, the read terminates after extracting the `delim` character.

If `len - 1` characters have been read and the next character is the `delim` character, it is not extracted.

After the read terminates, whether or not an error occurred, a null character is stored in the buffer following the last character read from the `istream` object.

If the `delim` parameter is not specified, the new-line character is assumed.

**Results:** The `getline` public member function returns a reference to the `istream` object. If end-of-file is encountered as the first character, `ios::eofbit` is set in the error state in the inherited `ios` object. If end-of-file is encountered before `len` characters are transferred or the `delim` character is reached, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `istream::get`, `read`, `operator >>`

## ***istream::ignore()***

---

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::ignore( int num = 1, int delim = EOF );
```

**Semantics:** The `ignore` public member function extracts and discards up to *num* characters from the `istream` object. If the *num* parameter is not specified, the `ignore` public member function extracts and discards one character. If the *delim* parameter is not `EOF` and it is encountered before *num* characters have been extracted, the extraction ceases after discarding the delimiting character. The extraction stops if end-of-file is encountered.

If the *num* parameter is specified as a negative number, no limit is imposed on the number of characters extracted and discarded. The operation continues until the delimiting character is found and discarded, or until end-of-file. This behavior is a WATCOM extension.

**Results:** The `ignore` public member function returns a reference to the `istream` object. If end-of-file is encountered as the first character, `ios::eofbit` is set in the error state in the inherited `ios` object.

**See Also:** `istream::eatwhite`



**Synopsis:**

```
#include <iostream.h>
public:
int istream::ipfx( int noskipws = 0 );
```

**Semantics:** The `ipfx` public member function is a prefix function executed before each of the formatted and unformatted read operations. If any bits are set in `ios::iostate`, the `ipfx` public member function immediately returns 0, indicating that the prefix function failed. Failure in the prefix function causes the input operation to fail.

If the `noskipws` parameter is 0 or unspecified and the `ios::skipws` bit is on in `ios::fmtflags`, whitespace characters are discarded and the `istream` object is positioned so that the next character read is the first character after the discarded whitespace. Otherwise, no whitespace skipping takes place.

The formatted input functions that read specific types of objects (such as integers and floating-point numbers) call the `ipfx` public member function with the `noskipws` parameter set to zero, allowing leading whitespaces to be discarded if the `ios::skipws` bit is on in `ios::fmtflags`. The unformatted input functions that read characters without interpretation call the `ipfx` public member function with a the `noskipws` parameter set to 1 so that no whitespace characters are discarded.

If the `istream` object is tied to an output stream, the output stream is flushed.

**Results:** If the `istream` object is not in an error state in the inherited `ios` object when the above processing is completed, the `ipfx` public member function returns a non-zero value to indicate success. Otherwise, zero is returned to indicate failure.

**See Also:** `istream::isfx`

## *istream::isfx()*

---

**Synopsis:**

```
#include <iostream.h>
public:
void istream::isfx();
```

**Semantics:** The `isfx` public member function is a suffix function executed just before the end of each of the formatted and unformatted read operations.

As currently implemented, the `isfx` public member function does not do anything.

**See Also:** `istream::ipfx`

**Synopsis:**

```
#include <iostream.h>
protected:
istream::istream();
```

**Semantics:** This form of the protected `istream` constructor creates an `istream` object without an associated `streambuf` object.

This form of the protected `istream` constructor is only used implicitly by the compiler when it generates a constructor for a derived class.

**Results:** This form of the protected `istream` constructor creates an initialized `istream` object. `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~istream`

## ***istream::istream()***

---

**Synopsis:**

```
#include <iostream.h>
public:
    istream::istream( istream const &istrm );
```

**Semantics:** This form of the public `istream` constructor creates an `istream` object associated with the `streambuf` object currently associated with the *istrm* parameter. The `istream` object is initialized and will use the *istrm* `streambuf` object for subsequent operations. *istrm* will continue to use the `streambuf` object.

**Results:** This form of the public `istream` constructor creates an initialized `istream` object. If there is no `streambuf` object currently associated with the *istrm* parameter, `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~istream`

**Synopsis:**

```
#include <iostream.h>
public:
istream::istream( streambuf *sb );
```

**Semantics:** This form of the public `istream` constructor creates an `istream` object with an associated `streambuf` object specified by the `sb` parameter.

This function is likely to be used for the creation of an `istream` object that is associated with the same `streambuf` object as another `istream` object.

**Results:** This form of the public `istream` constructor creates an initialized `istream` object. If the `sb` parameter is `NULL`, `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~istream`

## *istream::~~istream()*

---

**Synopsis:**

```
#include <iostream.h>
public:
virtual istream::~~istream();
```

**Semantics:** The public virtual `~istream` destructor does not do anything explicit. The `ios` destructor is called for that portion of the `istream` object. The call to the public virtual `~istream` destructor is inserted implicitly by the compiler at the point where the `istream` object goes out of scope.

**Results:** The `istream` object is destroyed.

**See Also:** `istream`

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::operator =( streambuf *sb );
```

**Semantics:** This form of the `operator =` public member function is used to associate a `streambuf` object, specified by the `sb` parameter, with an existing `istream` object. The `istream` object is initialized and will use the specified `streambuf` object for subsequent operations.

**Results:** This form of the `operator =` public member function returns a reference to the `istream` object that is the target of the assignment. If the `sb` parameter is `NULL`, `ios::badbit` is set in the error state in the inherited `ios` object.

## ***istream::operator =()***

---

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::operator =( istream const &istrm );
```

**Semantics:** This form of the `operator =` public member function is used to associate the `istream` object with the `streambuf` object currently associated with the `istrm` parameter. The `istream` object is initialized and will use the `istrm`'s `streambuf` object for subsequent operations. The `istrm` object will continue to use the `streambuf` object.

**Results:** This form of the `operator =` public member function returns a reference to the `istream` object that is the target of the assignment. If there is no `streambuf` object currently associated with the `istrm` parameter, `ios::badbit` is set in the error state in the inherited `ios` object.



**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::operator >>( char *buf );
istream &istream::operator >>( signed char *buf );
istream &istream::operator >>( unsigned char *buf );
```

**Semantics:** These forms of the operator `>>` public member function perform a formatted read of characters from the `istream` object and place them in the buffer specified by the *buf* parameter. Characters are read until a whitespace character is found or the maximum size has been read. If a whitespace character is found, it is not transferred to the buffer and remains in the `istream` object.

If a non-zero *format width* has been specified, it is interpreted as the maximum number of characters that may be placed in *buf*. No more than *format width*-1 characters are read from the `istream` object and transferred to *buf*. If *format width* is zero, characters are transferred until a whitespace character is found.

Since these forms of the operator `>>` public member function use *format width*, it is reset to zero after each use. It must be set before each input operation that requires a non-zero *format width*.

A null character is added following the last transferred character, even if the transfer fails because of an error.

**Results:** These forms of the operator `>>` public member function return a reference to the `istream` object so that further extraction operations may be specified in the same statement. If no characters are transferred to *buf*, `ios::failbit` is set in the error state in the inherited `ios` object. If the first character read yielded end-of-file, `ios::eofbit` is set in the error state in the inherited `ios` object.

**See Also:** `istream::get`, `getline`, `read`

## *istream::operator >>()*

---

**Synopsis:**

```
#include <iostream.h>
public:
    istream &istream::operator >>( char &ch );
    istream &istream::operator >>( signed char &ch );
    istream &istream::operator >>( unsigned char &ch );
```

**Semantics:** These forms of the operator `>>` public member function perform a formatted read of a single character from the `istream` object and place it in the `ch` parameter.

**Results:** These forms of the operator `>>` public member function return a reference to the `istream` object so that further extraction operations may be specified in the same statement. If the character read yielded end-of-file, `ios::eofbit` is set in the error state in the inherited `ios` object.

**See Also:** `istream::get`

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::operator >>( signed int &num );
istream &istream::operator >>( unsigned int &num );
istream &istream::operator >>( signed long &num );
istream &istream::operator >>( unsigned long &num );
istream &istream::operator >>( signed short &num );
istream &istream::operator >>( unsigned short &num );
```

**Semantics:** These forms the `operator >>` public member function perform a formatted read of an integral value from the `istream` object and place it in the `num` parameter.

The number may be preceded by a + or - sign.

If `ios::dec` is the only bit set in the `ios::basefield` bits of `ios::fmtflags`, the number is interpreted as a decimal (base 10) integer, composed of the digits 0123456789.

If `ios::oct` is the only bit set in the `ios::basefield` bits of `ios::fmtflags`, the number is interpreted as an octal (base 8) integer, composed of the digits 01234567.

If `ios::hex` is the only bit set in the `ios::basefield` bits of `ios::fmtflags`, the number is interpreted as a hexadecimal (base 16) integer, composed of the digits 0123456789 and the letters `abcdef` or `ABCDEF`.

If no bits are set in the `ios::basefield` bits of `ios::fmtflags`, the operator looks for a prefix to determine the base of the number. If the first two characters are `0x` or `0X`, the number is interpreted as a hexadecimal number. If the first character is a `0` (and the second is not an `x` or `X`), the number is interpreted as an octal integer. Otherwise, no prefix is expected and the number is interpreted as a decimal integer.

If more than one bit is set in the `ios::basefield` bits of `ios::fmtflags`, the number is interpreted as a decimal integer.

**Results:** These forms of the `operator >>` public member function return a reference to the `istream` object so that further extraction operations may be specified in the same statement. If end-of-file is encountered as the first character, `ios::eofbit` is set in the error state in the inherited `ios` object. If an overflow occurs while converting to the required integer type, the `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `ios::fmtflags`

## *istream::operator >>()*

---

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::operator >>( float &num );
istream &istream::operator >>( double &num );
istream &istream::operator >>( long double &num );
```

**Semantics:** These forms of the operator `>>` public member function perform a formatted read of a floating-point value from the `istream` object and place it in the *num* parameter.

The floating-point value may be specified in any form that is acceptable to the C++ compiler.

**Results:** These forms of the operator `>>` public member function return a reference to the `istream` object so that further extraction operations may be specified in the same statement. If end-of-file is encountered as the first character, `ios::eofbit` is set in the error state in the inherited `ios` object. If an overflow occurs while converting to the required type, the `ios::failbit` is set in the error state in the inherited `ios` object.

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::operator >>( streambuf &sb );
```

**Semantics:** This form of the `operator >>` public member function transfers all the characters from the `istream` object into the `sb` parameter. Reading continues until end-of-file is encountered.

**Results:** This form of the `operator >>` public member function return a reference to the `istream` object so that further extraction operations may be specified in the same statement.

## *istream::operator >>()*

---

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::operator >>( istream &(*fn)( istream & ) );
istream &istream::operator >>( ios &(*fn)( ios & ) );
```

**Semantics:** These forms of the `operator >>` public member function are used to implement the non-parameterized manipulators for the `istream` class. The function specified by the *fn* parameter is called with the `istream` object as its parameter.

**Results:** These forms of the `operator >>` public member function return a reference to the `istream` object so that further extraction operations may be specified in the same statement.

**Synopsis:**

```
#include <iostream.h>
public:
int istream::peek();
```

**Semantics:** The `peek` public member function looks up the next character to be extracted from the `istream` object, without extracting the character.

**Results:** The `peek` public member function returns the next character to be extracted from the `istream` object. If the `istream` object is positioned at end-of-file, EOF is returned.

**See Also:** `istream::get`

## ***istream::putback()***

---

**Synopsis:**

```
#include <iostream.h>
public:
    istream &istream::putback( char ch );
```

**Semantics:** The `putback` public member function attempts to put the extracted character specified by the `ch` parameter back into the `istream` object. The `ch` character must be the same as the character before the current position of the `istream` object, usually the last character extracted from the stream. If it is not the same character, the result of the next character extraction is undefined.

The number of characters that can be put back is defined by the `istream` object, but is usually at least 4. Depending on the status of the buffers used for input, it may be possible to put back more than 4 characters.

**Results:** The `putback` public member function returns a reference to the `istream` object. If the `putback` public member function is unable to put back the `ch` parameter, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `istream::get`



**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::read( char *buf, int len );
istream &istream::read( signed char *buf, int len );
istream &istream::read( unsigned char *buf, int len );
```

**Semantics:** The `read` public member function performs an unformatted read of at most *len* characters from the `istream` object and stores them in the memory locations starting at *buf*. If end-of-file is encountered before *len* characters have been transferred, the transfer stops and `ios::failbit` is set in the error state in the inherited `ios` object.

The number of characters extracted can be determined with the `gcount` member function.

**Results:** The `read` public member function returns a reference to the `istream` object. If end-of-file is encountered as the first character, `ios::eofbit` is set in the error state in the inherited `ios` object. If end-of-file is encountered before *len* characters are transferred, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `istream::gcount`, `get`, `getline`

## *istream::seekg()*

---

**Synopsis:**

```
#include <iostream.h>
public:
    istream &istream::seekg( streampos pos );
```

**Semantics:** The `seekg` public member function positions the `istream` object to the position specified by the `pos` parameter so that the next input operation commences from that position.

**Results:** The `seekg` public member function returns a reference to the `istream` object. If the seek operation fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `istream::tellg`, `ostream::tellp`, `ostream::seekp`

**Synopsis:**

```
#include <iostream.h>
public:
istream &istream::seekg( streamoff offset, ios::seekdir dir );
```

**Semantics:** The `seekg` public member function positions the `istream` object to the specified position so that the next input operation commences from that position.

The *dir* parameter may be `ios::beg`, `ios::cur`, or `ios::end` and is interpreted in conjunction with the *offset* parameter as follows:

`ios::beg` the *offset* is relative to the start and should be a positive value.  
`ios::cur` the *offset* is relative to the current position and may be positive (seek towards end) or negative (seek towards start).  
`ios::end` the *offset* is relative to the end and should be a negative value.

If the *dir* parameter has any other value, or the *offset* parameter does not have an appropriate sign, the `seekg` public member function fails.

**Results:** The `seekg` public member function returns a reference to the `istream` object. If the seek operation fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `ostream::tellp`, `ostream::seekp`  
`istream::tellg`

## *istream::sync()*

---

**Synopsis:**

```
#include <iostream.h>
public:
int istream::sync();
```

**Semantics:** The `sync` public member function synchronizes the input buffer and the `istream` object with whatever source of characters is being used. The `sync` public member function uses the `streambuf` class's `sync` virtual member function to carry out the synchronization. The specific behavior is dependent on what type of `streambuf` derived object is associated with the `istream` object.

**Results:** The `sync` public member function returns `__NOT_EOF` on success, otherwise `EOF` is returned.

**Synopsis:**

```
#include <iostream.h>
public:
streampos istream::tellg();
```

**Semantics:** The `tellg` public member function determines the position in the `istream` object of the next character available for reading. The first character in an `istream` object is at offset zero.

**Results:** The `tellg` public member function returns the position of the next character available for reading.

**See Also:** `ostream::tellp`, `ostream::seekp`  
`istream::seekg`

## *istream*

---

**Declared:** `strstrea.h`

**Derived from:**

`strstreambase`, `istream`

The `istream` class is used to create and read from string stream objects.

The `istream` class provides little of its own functionality. Derived from the `strstreambase` and `istream` classes, its constructors and destructor provide simplified access to the appropriate equivalents in those base classes.

Of the available I/O stream classes, creating an `istream` object is the preferred method of performing read operations from a string stream.

**Public Member Functions**

The following member functions are declared in the public interface:

```
istream( char * );  
istream( signed char * );  
istream( unsigned char * );  
istream( char *, int );  
istream( signed char *, int );  
istream( unsigned char *, int );  
~istream();
```

**See Also:** `istream`, `ostrstream`, `strstream`, `strstreambase`

**Synopsis:**

```
#include <strstrea.h>
public:
    istream::istream( char *str );
    istream::istream( signed char *str );
    istream::istream( unsigned char *str );
```

**Semantics:** This form of the public `istream` constructor creates an `istream` object consisting of the null terminated C string specified by the `str` parameter. The inherited `istream` member functions can be used to read from the `istream` object.

**Results:** This form of the public `istream` constructor creates an initialized `istream` object.

**See Also:** `~istream`

## ***istream::istream()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
    istream::istream( char *str, int len );
    istream::istream( signed char *str, int len );
    istream::istream( unsigned char *str, int len );
```

**Semantics:** This form of the public `istream` constructor creates an `istream` object consisting of the characters starting at `str` and ending at `str + len - 1`. The inherited `istream` member functions can be used to read from the `istream` object.

**Results:** This form of the public `istream` constructor creates an initialized `istream` object.

**See Also:** `~istream`



**Synopsis:** `#include <strstrea.h>`  
`public:`  
`istream::~~istream();`

**Semantics:** The public `~istream` destructor does not do anything explicit. The call to the public `~istream` destructor is inserted implicitly by the compiler at the point where the `istream` object goes out of scope.

**Results:** The `istream` object is destroyed.

**See Also:** `istream`

## Manipulators

---

**Declared:** `iostream.h` and `iomanip.h`

Manipulators are designed to be inserted into or extracted from a stream. Manipulators come in two forms, non-parameterized and parameterized. The non-parameterized manipulators are simpler and are declared in `<iostream.h>`. The parameterized manipulators require more complexity and are declared in `<iomanip.h>`.

`<iomanip.h>` defines two macros `SMANIP_define` and `SMANIP_make` to implement parameterized manipulators. The workings of the `SMANIP_define` and `SMANIP_make` macros are disclosed in the header file and are not discussed here.

### Non-parameterized Manipulators

The following non-parameterized manipulators are declared in `<iostream.h>`:

```
ios &dec( ios & );
ios &hex( ios & );
ios &oct( ios & );
istream &ws( istream & );
ostream &endl( ostream & );
ostream &ends( ostream & );
ostream &flush( ostream & );
```

### Parameterized Manipulators

The following parameterized manipulators are declared in `<iomanip.h>`:

```
SMANIP_define( long ) resetiosflags( long );
SMANIP_define( int ) setbase( int );
SMANIP_define( int ) setfill( int );
SMANIP_define( long ) setiosflags( long );
SMANIP_define( int ) setprecision( int );
SMANIP_define( int ) setw( int );
SMANIP_define( int ) setwidth( int );
```

**Synopsis:** `#include <iostream.h>`  
`ios &dec( ios &strm );`

**Semantics:** The `dec` manipulator sets the `ios::basefield` bits for decimal formatting in `ios::fmtflags` in the *strm* `ios` object.

**See Also:** `ios::fmtflags`

## ***manipulator endl()***

---

**Synopsis:** `#include <iostream.h>`  
`ostream &endl( ostream &ostrm );`

**Semantics:** The `endl` manipulator writes a new-line character to the stream specified by the *ostrm* parameter and performs a flush.

**See Also:** `ostream::flush`

**Synopsis:** `#include <iostream.h>`  
`ostream &ends( ostream &ostrm );`

**Semantics:** The ends manipulator writes a null character to the stream specified by the *ostrm* parameter.

## ***manipulator flush()***

---

**Synopsis:** `#include <iostream.h>`  
`ostream &flush( ostream &ostrm );`

**Semantics:** The `flush` manipulator flushes the stream specified by the `ostrm` parameter. The flush is performed in the same manner as the `flush` member function.

**See Also:** `ostream::flush`

**Synopsis:** `#include <iostream.h>`  
`ios &hex( ios &strm );`

**Semantics:** The hex manipulator sets the `ios::basefield` bits for hexadecimal formatting in `ios::fmtflags` in the *strm* ios object.

**See Also:** `ios::fmtflags`

## ***manipulator oct()***

---

**Synopsis:** `#include <iostream.h>`  
`ios &oct( ios &strm );`

**Semantics:** The `oct` manipulator sets the `ios::basefield` bits for octal formatting in `ios::fmtflags` in the *strm* `ios` object.

**See Also:** `ios::fmtflags`



**Synopsis:** `#include <iomanip.h>`  
`SMANIP_define( long ) resetiosflags( long flags )`

**Semantics:** The `resetiosflags` manipulator turns off the bits in `ios::fmtflags` that correspond to the bits that are on in the *flags* parameter. No other bits are affected.

**See Also:** `ios::flags`, `ios::fmtflags`, `ios::setf`, `ios::unsetf`

## ***manipulator setbase()***

---

**Synopsis:** `#include <iomanip.h>`  
`SMANIP_define( int ) setbase( int base );`

**Semantics:** The `setbase` manipulator sets the `ios::basefield` bits in `ios::fmtflags` to the value specified by the *base* parameter within the stream that the `setbase` manipulator is operating upon.

**See Also:** `ios::fmtflags`

**Synopsis:** `#include <iomanip.h>`  
`SMANIP_define( int ) setfill( int fill )`

**Semantics:** The `setfill` manipulator sets the *fill character* to the value specified by the *fill* parameter within the stream that the `setfill` manipulator is operating upon.

**See Also:** `ios::fill`

## ***manipulator setiosflags()***

---

**Synopsis:** `#include <iomanip.h>`  
`SMANIP_define( long ) setiosflags( long flags );`

**Semantics:** The `setiosflags` manipulator turns on the bits in `ios::fmtflags` that correspond to the bits that are on in the *flags* parameter. No other bits are affected.

**See Also:** `ios::flags`, `ios::fmtflags`, `ios::setf`, `ios::unsetf`

**Synopsis:** `#include <iomanip.h>`  
`SMANIP_define( int ) setprecision( int prec );`

**Semantics:** The `setprecision` manipulator sets the *format precision* to the value specified by the *prec* parameter within the stream that the `setprecision` manipulator is operating upon.

**See Also:** `ios::precision`

## ***manipulator setw()***

---

**Synopsis:** `#include <iomanip.h>`  
`SMANIP_define( int ) setw( int wid );`

**Semantics:** The `setw` manipulator sets the *format width* to the value specified by the *wid* parameter within the stream that the `setw` manipulator is operating upon.

**See Also:** `ios::width`, manipulator `setw`

**Synopsis:** `#include <iomanip.h>`  
`SMANIP_define( int ) setw( int wid );`

**Semantics:** The `setw` manipulator sets the *format width* to the value specified by the *wid* parameter within the stream that the `setw` manipulator is operating upon.

This function is a WATCOM extension.

**See Also:** `ios::width`, manipulator `setw`

## ***manipulator ws()***

---

**Synopsis:**

```
#include <iostream.h>
istream &ws( istream &istrm );
```

**Semantics:** The `ws` manipulator extracts and discards whitespace characters from the `istrm` parameter, leaving the stream positioned at the next non-whitespace character.

The `ws` manipulator is needed particularly when the `ios::skipws` bit is not set in `ios::fmtflags` in the `istrm` object. In this case, whitespace characters must be explicitly removed from the stream, since the formatted input operations will not automatically remove them.

**See Also:** `istream::eatwhite`, `istream::ignore`



**Declared:** `fstream.h`

**Derived from:**

`fstreambase`, `ostream`

The `ofstream` class is used to create new files or access existing files for writing. The file can be opened and closed, and write and seek operations can be performed.

The `ofstream` class provides very little of its own functionality. Derived from both the `fstreambase` and `ostream` classes, its constructors, destructor and member function provide simplified access to the appropriate equivalents in those base classes.

Of the available I/O stream classes, creating an `ofstream` object is the preferred method of accessing a file for output operations.

**Public Member Functions**

The following public member functions are declared:

```
ofstream();  
ofstream( char const *,  
ios::openmode = ios::out,  
int = filebuf::openprot );  
ofstream( filedesc );  
ofstream( filedesc, char *, int );  
~ofstream();  
void open( char const *,  
ios::openmode = ios::out,  
int = filebuf::openprot );
```

**See Also:** `fstream`, `fstreambase`, `ifstream`, `ostream`

## ***ofstream::ofstream()***

---

**Synopsis:**

```
#include <fstream.h>
public:
ofstream::ofstream();
```

**Semantics:** This form of the public `ofstream` constructor creates an `ofstream` object that is not connected to a file. The `open` or `attach` member functions should be used to connect the `ofstream` object to a file.

**Results:** The public `ofstream` constructor produces an `ofstream` object that is not connected to a file.

**See Also:** `~ofstream`

**Synopsis:**

```
#include <fstream.h>
public:
ofstream::ofstream( const char *name,
ios::openmode mode = ios::out,
int prot = filebuf::openprot );
```

**Semantics:** This form of the public `ofstream` constructor creates an `ofstream` object that is connected to the file specified by the *name* parameter, using the specified *mode* and *prot* parameters. The connection is made via the C library `open` function.

**Results:** The public `ofstream` constructor produces an `ofstream` object that is connected to the file specified by *name*. If the `open` fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `~ofstream`, `open`, `fstreambase::close`, `openmode`, `openprot`

## ***ofstream::ofstream()***

---

**Synopsis:**

```
#include <fstream.h>
public:
ofstream::ofstream( filedesc hdl );
```

**Semantics:** This form of the public `ofstream` constructor creates an `ofstream` object that is attached to the file specified by the *hdl* parameter.

**Results:** The public `ofstream` constructor produces an `ofstream` object that is attached to *hdl*. If the attach fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object.

**See Also:** `~ofstream`, `fstreambase::attach`, `fstreambase::fd`

**Synopsis:**

```
#include <fstream.h>
public:
ofstream::ofstream( filedesc hdl, char *buf, int len );
```

**Semantics:** This form of the public `ofstream` constructor creates an `ofstream` object that is connected to the file specified by the *hdl* parameter. The buffer specified by the *buf* and *len* parameters is offered to the associated `filebuf` object via the `setbuf` member function. If the *buf* parameter is `NULL` or the *len* is less than or equal to zero, the `filebuf` is unbuffered, so that each read or write operation reads or writes a single character at a time.

**Results:** The public `ofstream` constructor produces an `ofstream` object that is attached to *hdl*. If the connection to *hdl* fails, `ios::failbit` and `ios::badbit` are set in the error state in the inherited `ios` object. If the `setbuf` fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `~ofstream`, `fstreambase::attach`, `fstreambase::fd`, `fstreambase::setbuf`

## ***ofstream::~~ofstream()***

---

**Synopsis:**

```
#include <fstream.h>
public:
ofstream::~~ofstream();
```

**Semantics:** The public `~ofstream` destructor does not do anything explicit. The call to the public `~ofstream` destructor is inserted implicitly by the compiler at the point where the `ofstream` object goes out of scope.

**Results:** The public `~ofstream` destructor destroys the `ofstream` object.

**See Also:** `ofstream`

**Synopsis:**

```
#include <fstream.h>
public:
void ofstream::open( const char *name,
ios::openmode mode = ios::out,
int prot = filebuf::openprot );
```

**Semantics:** The `open` public member function connects the `ofstream` object to the file specified by the *name* parameter, using the specified *mode* and *prot* parameters. The *mode* parameter is optional and usually is not specified unless additional bits (such as `ios::binary` or `ios::text`) are to be specified. The connection is made via the C library `open` function.

**Results:** If the `open` fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `ofstream`, `openmode`, `openprot`, `fstreambase::attach`,  
`fstreambase::close`, `fstreambase::fd`, `fstreambase::is_open`

## ***ostream***

---

**Declared:** `iostream.h`

**Derived from:**  
`ios`

**Derived by:** `iostream, ofstream, ostrstream`

The `ostream` class supports writing characters to a class derived from the `streambuf` class, and provides formatted conversion of types (such as integers and floating-point numbers) into characters. The class derived from the `streambuf` class provides the methods for communicating with the external device (screen, disk), while the `ostream` class provides the conversion of the types into characters.

Generally, `ostream` objects won't be explicitly created by a program, since there is no mechanism at this level to open a device. The only default `ostream` objects in a program are `cout`, `cerr`, and `clog` which write to the standard output and error devices (usually the screen).

The `ostream` class supports two basic concepts of output: formatted and unformatted. The overloaded operator `<<` member functions are called *inserters* and they provide the support for formatted output. The rest of the member functions deal with unformatted output, managing the state of the `ios` object and providing a friendlier interface to the associated `streambuf` object.

### **Protected Member Functions**

The following protected member functions are declared:

```
ostream();
```

### **Public Member Functions**

The following public member functions are declared:

```
ostream( ostream const & );  
ostream( streambuf * );  
virtual ~ostream();  
ostream &flush();  
int opfx();  
void osfx();  
ostream &put( char );  
ostream &put( signed char );  
ostream &put( unsigned char );  
ostream &seekp( streampos );  
ostream &seekp( streamoff, ios::seekdir );
```



```
streampos tellp();
ostream &write( char const *, int );
ostream &write( signed char const *, int );
ostream &write( unsigned char const *, int );
```

### **Public Member Operators**

The following public member operators are declared:

```
ostream &operator =( ostreambuf * );
ostream &operator =( ostream const & );
ostream &operator <<( char );
ostream &operator <<( signed char );
ostream &operator <<( unsigned char );
ostream &operator <<( signed short );
ostream &operator <<( unsigned short );
ostream &operator <<( signed int );
ostream &operator <<( unsigned int );
ostream &operator <<( signed long );
ostream &operator <<( unsigned long );
ostream &operator <<( float );
ostream &operator <<( double );
ostream &operator <<( long double );
ostream &operator <<( void * );
ostream &operator <<( ostreambuf & );
ostream &operator <<( char const * );
ostream &operator <<( signed char const * );
ostream &operator <<( unsigned char const * );
ostream &operator <<( ostream &(*) ( ostream & ) );
ostream &operator <<( ios &(*) ( ios & ) );
```

**See Also:** ios, ostream, istream

## ***ostream::flush()***

---

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::flush();
```

**Semantics:** The `flush` public member function causes the `ostream` object's buffers to be flushed, forcing the contents to be written to the actual device connected to the `ostream` object.

**Results:** The `flush` public member function returns a reference to the `ostream` object. On failure, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `ostream::osfx`

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator <<( char ch );
ostream &ostream::operator <<( signed char ch );
ostream &ostream::operator <<( unsigned char ch );
```

**Semantics:** These forms of the operator << public member function write the *ch* character into the ostream object.

**Results:** These forms of the operator << public member function return a reference to the ostream object so that further insertion operations may be specified in the same statement. ios::failbit is set in the error state in the inherited ios object if an error occurs.

**See Also:** ostream::put

## ***ostream::operator <<()***

---

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator <<( char const *str );
ostream &ostream::operator <<( signed char const *str );
ostream &ostream::operator <<( unsigned char const *str );
```

**Semantics:** These forms of the operator << public member function perform a formatted write of the contents of the C string specified by the *str* parameter to the *ostream* object. The characters from *str* are transferred up to, but not including the terminating null character.

**Results:** These forms of the operator << public member function return a reference to the *ostream* object so that further insertion operations may be specified in the same statement. `ios::failbit` is set in the error state in the inherited *ios* object if an error occurs.

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator <<( signed int num );
ostream &ostream::operator <<( unsigned int num );
ostream &ostream::operator <<( signed long num );
ostream &ostream::operator <<( unsigned long num );
ostream &ostream::operator <<( signed short num );
ostream &ostream::operator <<( unsigned short num );
```

**Semantics:** These forms of the operator << public member function perform a formatted write of the integral value specified by the *num* parameter to the *ostream* object. The integer value is converted to a string of characters which are written to the *ostream* object. *num* is converted to a base representation depending on the setting of the *ios::basefield* bits in *ios::fmtflags*. If the *ios::oct* bit is the only bit on, the conversion is to an octal (base 8) representation. If the *ios::hex* bit is the only bit on, the conversion is to a hexadecimal (base 16) representation. Otherwise, the conversion is to a decimal (base 10) representation.

For decimal conversions only, a sign may be written in front of the number. If the number is negative, a - minus sign is written. If the number is positive and the *ios::showpos* bit is on in *ios::fmtflags*, a + plus sign is written. No sign is written for a value of zero.

If the *ios::showbase* bit is on in *ios::fmtflags*, and the conversion is to octal or hexadecimal, the base indicator is written next. The base indicator for a conversion to octal is a zero. The base indicator for a conversion to hexadecimal is 0x or 0X, depending on the setting of the *ios::uppercase* bit in *ios::fmtflags*.

If the value being written is zero, the conversion is to octal, and the *ios::showbase* bit is on, nothing further is written since a single zero is sufficient.

The value of *num* is then converted to characters. For conversions to decimal, the magnitude of the number is converted to a string of decimal digits 0123456789. For conversions to octal, the number is treated as an unsigned quantity and converted to a string of octal digits 01234567. For conversions to hexadecimal, the number is treated as an unsigned quantity and converted to a string of hexadecimal digits 0123456789 and the letters abcdef or ABCDEF, depending on the setting of the *ios::uppercase* in *ios::fmtflags*. The string resulting from the conversion is then written to the *ostream* object.

If the *ios::internal* bit is set in *ios::fmtflags* and padding is required, the padding characters are written after the sign and/or base indicator (if present) and before the digits.

## *ostream::operator <<()*

---

**Results:** These forms of the operator `<<` public member function return a reference to the `ostream` object so that further insertion operations may be specified in the same statement. `ios::failbit` is set in the error state in the inherited `ios` object if an error occurs.

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator <<( float num );
ostream &ostream::operator <<( double num );
ostream &ostream::operator <<( long double num );
```

**Semantics:** These forms of the operator << public member function perform a formatted write of the floating-point value specified by the *num* parameter to the *ostream* object. The number is converted to either scientific (exponential) form or fixed-point form, depending on the setting of the *ios::floatfield* bits in *ios::fmtflags*. If *ios::scientific* is the only bit set, the conversion is to scientific form. If *ios::fixed* is the only bit set, the conversion is to fixed-point form. Otherwise (neither or both bits set), the value of the number determines the conversion used. If the exponent is less than -4 or is greater than or equal to the *format precision*, the scientific form is used. Otherwise, the fixed-point form is used.

Scientific form consists of a minus sign (for negative numbers), one digit, a decimal point, *format precision*-1 digits, an e or E (depending on the setting of the *ios::uppercase* bit), a minus sign (for negative exponents) or a plus sign (for zero or positive exponents), and two or three digits for the exponent. The digit before the decimal is not zero, unless the number is zero. If the *format precision* is zero (or one), no digits are written following the decimal point.

Fixed-point form consists of a minus sign (for negative numbers), one or more digits, a decimal point, and *format precision* digits.

If the *ios::showpoint* bit is not set in *ios::fmtflags*, trailing zeroes are trimmed after the decimal point (and before the exponent for scientific form), and if no digits remain after the decimal point, the decimal point is discarded as well.

If the *ios::internal* bit is set in *ios::fmtflags* and padding is required, the padding characters are written after the sign (if present) and before the digits.

**Results:** These forms of the operator << public member function return a reference to the *ostream* object so that further insertion operations may be specified in the same statement. *ios::failbit* is set in the error state in the inherited *ios* object if an error occurs.

## ***ostream::operator <<()***

---

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator <<( void *ptr );
```

**Semantics:** This form of the `operator <<` public member function performs a formatted write of the pointer value specified by the `ptr` parameter to the `ostream` object. The `ptr` parameter is converted to an implementation-defined string of characters and written to the `ostream` object. With the Open Watcom C++ implementation, the string starts with `0x` or `0X` (depending on the setting of the `ios::uppercase` bit), followed by 4 hexadecimal digits for 16-bit pointers and 8 hexadecimal digits for 32-bit pointers. Leading zeroes are added to ensure the correct number of digits are written. For far pointers, 4 additional hexadecimal digits and a colon are inserted immediately after the `0x` prefix.

**Results:** This form of the `operator <<` public member function returns a reference to the `ostream` object so that further insertion operations may be specified in the same statement. `ios::failbit` is set in the error state in the inherited `ios` object if an error occurs during the write.



**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator <<( ostreambuf &sb );
```

**Semantics:** This form of the `operator <<` public member function transfers the contents of the `sb` `ostreambuf` object to the `ostream` object. Reading from the `ostreambuf` object stops when the read fails. No padding with the *fill character* takes place on output to the `ostream` object.

**Results:** This form of the `operator <<` public member function returns a reference to the `ostream` object so that further insertion operations may be specified in the same statement. `ios::failbit` is set in the error state in the inherited `ios` object if an error occurs.

## ***ostream::operator <<()***

---

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator <<( ostream &(*fn)( ostream & ) );
ostream &ostream::operator <<( ios &(*fn)( ios & ) );
```

**Semantics:** These forms of the `operator <<` public member function are used to implement the non-parameterized manipulators for the `ostream` class. The function specified by the *fn* parameter is called with the `ostream` object as its parameter.

**Results:** These forms of the `operator <<` public member function return a reference to the `ostream` object so that further insertions operations may be specified in the same statement.

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator =( streambuf *sb );
```

**Semantics:** This form of the `operator =` public member function is used to associate a `streambuf` object, specified by the `sb` parameter, with an existing `ostream` object. The `ostream` object is initialized and will use the specified `streambuf` object for subsequent operations.

**Results:** This form of the `operator =` public member function returns a reference to the `ostream` object that is the target of the assignment. If the `sb` parameter is `NULL`, `ios::badbit` is set in the error state in the inherited `ios` object.

## ***ostream::operator =()***

---

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::operator =( const ostream &ostrm );
```

**Semantics:** This form of the `operator =` public member function is used to associate the `ostream` object with the `streambuf` object currently associated with the `ostrm` parameter. The `ostream` object is initialized and will use the `ostrm`'s `streambuf` object for subsequent operations. The `ostrm` object will continue to use the `streambuf` object.

**Results:** This form of the `operator =` public member function returns a reference to the `ostream` object that is the target of the assignment. If there is no `streambuf` object currently associated with the `ostrm` parameter, `ios::badbit` is set in the error state in the inherited `ios` object.

**Synopsis:**

```
#include <iostream.h>
public:
int ostream::opfx();
```

**Semantics:** If `opfx` public member function is a prefix function executed before each of the formatted and unformatted output operations. If any bits are set in `ios::iostate`, the `opfx` public member function immediately returns zero, indicating that the prefix function failed. Failure in the prefix function causes the output operation to fail.

If the `ostream` object is tied to another `ostream` object, the other `ostream` object is flushed.

**Results:** The `opfx` public member function returns a non-zero value on success, otherwise zero is returned.

**See Also:** `ostream::osfx`, `flush`, `ios::tie`

## ***ostream::osfx()***

---

**Synopsis:**

```
#include <iostream.h>
public:
void ostream::osfx();
```

**Semantics:** The `osfx` public member function is a suffix function executed at the end of each of the formatted and unformatted output operations.

If the `ios::unitbuf` bit is set in `ios::fmtflags`, the `flush` member function is called. If the `ios::stdio` bit is set in `ios::fmtflags`, the C library `fflush` function is invoked on the `stdout` and `stderr` file streams.

**See Also:** `ostream::osfx`, `flush`

**Synopsis:** `#include <iostream.h>`  
`protected:`  
`ostream::ostream();`

**Semantics:** This form of the protected `ostream` constructor creates an `ostream` object without an attached `streambuf` object.

This form of the protected `ostream` constructor is only used implicitly by the compiler when it generates a constructor for a derived class.

**Results:** This form of the protected `ostream` constructor creates an initialized `ostream` object. `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~ostream`

## ***ostream::ostream()***

---

**Synopsis:**

```
#include <iostream.h>
public:
ostream::ostream( ostream const &ostrm );
```

**Semantics:** This form of the public `ostream` constructor creates an `ostream` object associated with the `streambuf` object currently associated with the `ostrm` parameter. The `ostream` object is initialized and will use the `ostrm`'s `streambuf` object for subsequent operations. The `ostrm` object will continue to use the `streambuf` object.

**Results:** This form of the public `ostream` constructor creates an initialized `ostream` object. If there is no `streambuf` object currently associated with the `ostrm` parameter, `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~ostream`



**Synopsis:**

```
#include <iostream.h>
public:
ostream::ostream( streambuf *sb );
```

**Semantics:** This form of the public `ostream` constructor creates an `ostream` object with an associated `streambuf` object specified by the `sb` parameter.

This function is likely to be used for the creation of an `ostream` object that is associated with the same `streambuf` object as another `ostream` object.

**Results:** This form of the public `ostream` constructor creates an initialized `ostream` object. If the `sb` parameter is `NULL`, `ios::badbit` is set in the error state in the inherited `ios` object.

**See Also:** `~ostream`

## ***ostream::~~ostream()***

---

**Synopsis:**

```
#include <iostream.h>
public:
virtual ostream::~~ostream();
```

**Semantics:** The public virtual `~ostream` destructor does not do anything explicit. The `ios` destructor is called for that portion of the `ostream` object. The call to the public virtual `~ostream` destructor is inserted implicitly by the compiler at the point where the `ostream` object goes out of scope.

**Results:** The `ostream` object is destroyed.

**See Also:** `ostream`

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::put( char ch );
ostream &ostream::put( signed char ch );
ostream &ostream::put( unsigned char ch );
```

**Semantics:** These forms of the `put` public member function write the *ch* character to the `ostream` object.

**Results:** These forms of the `put` public member function return a reference to the `ostream` object. If an error occurs, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `ostream::operator <<`, `write`

## ***ostream::seekp()***

---

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::seekp( streampos pos );
```

**Semantics:** This from of the `seekp` public member function positions the `ostream` object to the position specified by the `pos` parameter so that the next output operation commences from that position.

The `pos` value is an absolute position within the stream. It may be obtained via a call to the `tellp` member function.

**Results:** This from of the `seekp` public member function returns a reference to the `ostream` object. If the seek operation fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `ostream::tellp`, `istream::tellg`, `istream::seekg`

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::seekp( streamoff offset, ios::seekdir dir );
```

**Semantics:** This from of the `seekp` public member function positions the `ostream` object to the specified position so that the next output operation commences from that position.

The *dir* parameter may be `ios::beg`, `ios::cur`, or `ios::end` and is interpreted in conjunction with the *offset* parameter as follows:

`ios::beg` the *offset* is relative to the start and should be a positive value.  
`ios::cur` the *offset* is relative to the current position and may be positive (seek towards end) or negative (seek towards start).  
`ios::end` the *offset* is relative to the end and should be a negative value.

If the *dir* parameter has any other value, or the *offset* parameter does not have an appropriate sign, the `seekp` public member function fails.

**Results:** This from of the `seekp` public member function returns a reference to the `ostream` object. If the seek operation fails, `ios::failbit` is set in the error state in the inherited `ios` object.

**See Also:** `ostream::tellp`, `istream::tellg`, `istream::seekg`

## ***ostream::tellp()***

---

**Synopsis:**

```
#include <iostream.h>
public:
streampos ostream::tellp();
```

**Semantics:** The `tellp` public member function returns the position in the `ostream` object at which the next character will be written. The first character in an `ostream` object is at offset zero.

**Results:** The `tellp` public member function returns the position in the `ostream` object at which the next character will be written.

**See Also:** `ostream::seekp`, `istream::tellg`, `istream::seekg`

**Synopsis:**

```
#include <iostream.h>
public:
ostream &ostream::write( char const *buf, int len );
ostream &ostream::write( signed char const *buf, int len );
ostream &ostream::write( unsigned char const *buf, int len );
```

**Semantics:** The `write` public member function performs an unformatted write of the characters specified by the *buf* and *len* parameters into the `ostream` object.

**Results:** These member functions return a reference to the `ostream` object. If an error occurs, `ios::failbit` is set in the error state in the inherited `ios` object.

## ***ostream***

---

**Declared:** `strstream.h`

**Derived from:**

`strstreambase`, `ostream`

The `ostream` class is used to create and write to string stream objects.

The `ostream` class provides little of its own functionality. Derived from the `strstreambase` and `ostream` classes, its constructors and destructor provide simplified access to the appropriate equivalents in those base classes. The member functions provide specialized access to the string stream object.

Of the available I/O stream classes, creating an `ostream` object is the preferred method of performing write operations to a string stream.

**Public Member Functions**

The following member functions are declared in the public interface:

```
ostream();  
ostream( char *, int, ios::openmode = ios::out );  
ostream( signed char *, int, ios::openmode = ios::out );  
ostream( unsigned char *, int, ios::openmode = ios::out );  
~ostream();  
int pcount() const;  
char *str();
```

**See Also:** `istream`, `ostream`, `ostream`, `strstreambase`



**Synopsis:**

```
#include <strstrea.h>
public:
ostream::ostream();
```

**Semantics:** This form of the public `ostream` constructor creates an empty `ostream` object. Dynamic allocation is used. The inherited stream member functions can be used to access the `ostream` object.

**Results:** This form of the public `ostream` constructor creates an initialized, empty `ostream` object.

**See Also:** `~ostream`

## ***ostream::ostream()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
ostream::ostream( char *str,
int len,
ios::openmode mode = ios::out );
ostream::ostream( signed char *str,
int len,
ios::openmode mode = ios::out );
ostream::ostream( unsigned char *str,
int len,
ios::openmode mode = ios::out );
```

**Semantics:** These forms of the public `ostream` constructor create an initialized `ostream` object. Dynamic allocation is not used. The buffer is specified by the *str* and *len* parameters. If the `ios::append` or `ios::atend` bits are set in the *mode* parameter, the *str* parameter is assumed to contain a C string terminated by a null character, and writing commences at the null character. Otherwise, writing commences at *str*.

**Results:** This form of the public `ostream` constructor creates an initialized `ostream` object.

**See Also:** `~ostream`

**Synopsis:** `#include <ostream.h>`  
`public:`  
`ostream::~ostream();`

**Semantics:** The public `~ostream` destructor does not do anything explicit. The call to the public `~ostream` destructor is inserted implicitly by the compiler at the point where the `ostream` object goes out of scope.

**Results:** The `ostream` object is destroyed.

**See Also:** `ostream`

## ***ostream::pcount()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
int ostream::pcount() const;
```

**Semantics:** The `pcount` public member function computes the number of characters that have been written to the `ostream` object. This value is particularly useful if the `ostream` object does not contain a C string (terminated by a null character), so that the number of characters cannot be determined with the C library `strlen` function. If the `ostream` object was created by appending to a C string in a static buffer, the length of the original string is included in the character count.

**Results:** The `pcount` public member function returns the number of characters contained in the `ostream` object.

**Synopsis:**

```
#include <strstream.h>
public:
char *ostream::str();
```

**Semantics:** The `str` public member function creates a pointer to the buffer being used by the `ostream` object. If the `ostream` object was created without dynamic allocation (static mode), the pointer is the same as the buffer pointer passed in the constructor.

For `ostream` objects using dynamic allocation, the `str` public member function makes an implicit call to the `strstreambuf::freeze` member function. If nothing has been written to the `ostream` object, the returned pointer will be `NULL`.

Note that the buffer does not necessarily end with a null character. If the pointer returned by the `str` public member function is to be interpreted as a C string, it is the program's responsibility to ensure that the null character is present.

**Results:** The `str` public member function returns a pointer to the buffer being used by the `ostream` object.

**Declared:** `stdiobuf.h`

**Derived from:**

`streambuf`

The `stdiobuf` class specializes the `streambuf` class and is used to implement the standard input/output buffering required for the `cin`, `cout`, `cerr` and `clog` predefined objects.

The `stdiobuf` class behaves in a similar way to the `filebuf` class, but does not need to switch between the *get area* and *put area*, since no `stdiobuf` object can be created for both reading and writing. When the *get area* is empty and a read is done, the `underflow` virtual member function reads more characters and fills the *get area* again. When the *put area* is full and a write is done, the `overflow` virtual member function writes the characters and makes the *put area* empty again.

C++ programmers who wish to use the standard input/output streams without deriving new objects do not need to explicitly create or use a `stdiobuf` object.

### **Public Member Functions**

The following member functions are declared in the public interface:

```
stdiobuf();  
stdiobuf( FILE * );  
~stdiobuf();  
virtual int overflow( int = EOF );  
virtual int underflow();  
virtual int sync();
```

**See Also:** `streambuf`, `ios`

**Synopsis:**

```
#include <stdiobuf.h>
public:
virtual int stdiobuf::overflow( int ch = EOF );
```

**Semantics:** The `overflow` public virtual member function provides the output communication to the standard output and standard error devices to which the `stdiobuf` object is connected. Member functions in the `streambuf` class call the `overflow` public virtual member function for the derived class when the *put area* is full.

The `overflow` public virtual member function performs the following steps:

1. If no buffer is present, a buffer is allocated with the `streambuf::allocate` member function, which may call the `doallocate` virtual member function. The *put area* is then set up. If, after calling `streambuf::allocate`, no buffer is present, the `stdiobuf` object is unbuffered and *ch* (if not `EOF`) is written directly to the file without buffering, and no further action is taken.
2. If the *get area* is present, it is flushed with a call to the `sync` virtual member function. Note that the *get area* won't be present if a buffer was set up in step 1.
3. If *ch* is not `EOF`, it is added to the *put area*, if possible.
4. Any characters in the *put area* are written to the file.
5. The *put area* pointers are updated to reflect the new state of the *put area*. If the write did not complete, the unwritten portion of the *put area* is still present. If the *put area* was full before the write, *ch* (if not `EOF`) is placed at the start of the *put area*. Otherwise, the *put area* is empty.

**Results:** The `overflow` public virtual member function returns `__NOT_EOF` on success, otherwise `EOF` is returned.

**See Also:** `stdiobuf::underflow`, `streambuf::overflow`

## ***stdiobuf::stdiobuf()***

---

**Synopsis:**

```
#include <stdiobuf.h>
public:
stdiobuf::stdiobuf();
```

**Semantics:** This form of the public `stdiobuf` constructor creates a `stdiobuf` object that is initialized but not yet connected to a file.

**Results:** This form of the public `stdiobuf` constructor creates a `stdiobuf` object.

**See Also:** `~stdiobuf`



**Synopsis:**

```
#include <stdiobuf.h>
public:
stdiobuf::stdiobuf( FILE *fptr );
```

**Semantics:** This form of the public `stdiobuf` constructor creates a `stdiobuf` object that is initialized and connected to a C library `FILE` stream. Usually, one of `stdin`, `stdout` or `stderr` is specified for the *fptr* parameter.

**Results:** This form of the public `stdiobuf` constructor creates a `stdiobuf` object that is initialized and connected to a C library `FILE` stream.

**See Also:** `~stdiobuf`

## ***stdiobuf::~~stdiobuf()***

---

**Synopsis:**

```
#include <stdiobuf.h>
public:
stdiobuf::~~stdiobuf();
```

**Semantics:** The public `~stdiobuf` destructor does not do anything explicit. The `streambuf` destructor is called for that portion of the `stdiobuf` object. The call to the public `~stdiobuf` destructor is inserted implicitly by the compiler at the point where the `stdiobuf` object goes out of scope.

**Results:** The `stdiobuf` object is destroyed.

**See Also:** `stdiobuf`

**Synopsis:**

```
#include <stdiobuf.h>
public:
virtual int stdiobuf::sync();
```

**Semantics:** The `sync` public virtual member function synchronizes the `stdiobuf` object with the associated device. If the *put area* contains characters, it is flushed. If the *get area* contains buffered characters, the `sync` public virtual member function fails.

**Results:** The `sync` public virtual member function returns `__NOT_EOF` on success, otherwise, `EOF` is returned.

**See Also:** `streambuf::sync`

## ***stdiobuf::underflow()***

---

**Synopsis:**

```
#include <stdiobuf.h>
public:
virtual int stdiobuf::underflow();
```

**Semantics:** The `underflow` public virtual member function provides the input communication from the standard input device to which the `stdiobuf` object is connected. Member functions in the `streambuf` class call the `underflow` public virtual member function for the derived class when the *get area* is empty.

The `underflow` public virtual member function performs the following steps:

1. If no *reserve area* is present, a buffer is allocated with the `streambuf::allocate` member function, which may call the `doallocate` virtual member function. If, after calling `allocate`, no *reserve area* is present, the `stdiobuf` object is unbuffered and a one-character *reserve area* (plus putback area) is set up to do unbuffered input. This buffer is embedded in the `stdiobuf` object. The *get area* is set up as empty.
2. The unused part of the *get area* is used to read characters from the file connected to the `stdiobuf` object. The *get area* pointers are then set up to reflect the new *get area*.

**Results:** The `underflow` public virtual member function returns the first unread character of the *get area*, on success, otherwise EOF is returned. Note that the *get pointer* is not advanced on success.

**See Also:** `stdiobuf::overflow`, `streambuf::underflow`

**Declared:** `streambu.h`

**Derived by:** `filebuf`, `stdiobuf`, `strstreambuf`

The `streambuf` class is responsible for maintaining the buffer used to create an efficient implementation of the stream classes. Through its pure virtual functions, it is also responsible for the actual communication with the device associated with the stream.

The `streambuf` class is abstract, due to the presence of pure virtual member functions. Abstract classes may not be instantiated, only inherited. Hence, `streambuf` objects will not be created by user programs.

Stream objects maintain a pointer to an associated `streambuf` object and present the interface that the user deals with most often. Whenever a stream member function wishes to read or write characters, it uses the `rdbuf` member function to access the associated `streambuf` object and its member functions. Through judicious use of inline functions, most reads and writes of characters access the buffer directly without even doing a function call. Whenever the buffer gets filled (writing) or exhausted (reading), these inline functions invoke the function required to rectify the situation so that the proper action can take place.

A `streambuf` object can be unbuffered, but most often has one buffer which can be used for both input and output operations. The buffer (called the *reserve area*) is divided into two areas, called the *get area* and the *put area*. For a `streambuf` object being used exclusively to write, the *get area* is empty or not present. Likewise, a `streambuf` object being used exclusively for reading has an empty or non-existent *put area*.

The use of the *get area* and *put area* differs among the various classes derived from the `streambuf` class.

The `filebuf` class allows only the *get area* or the *put area*, but not both, to be active at a time. This follows from the capability of files opened for both reading and writing to have operations of each type performed at arbitrary locations in the file. When writing is occurring, the characters are buffered in the *put area*. If a seek or read operation is done, the *put area* must be flushed before the next operation in order to ensure that the characters are written to the proper location in the file. Similarly, if reading is occurring, characters are buffered in the *get area*. If a write operation is done, the *get area* must be flushed and synchronized before the write operation in order to ensure the write occurs at the proper location in the file. If a seek operation is done, the *get area* does not have to be synchronized, but is discarded. When the *get area* is empty and a read is done, the `underflow` virtual member function reads more characters and fills the *get area* again. When the *put area* is full and a write is done, the `overflow` virtual member function writes the characters and makes the *put area* empty again.

The `std::streambuf` class behaves in a similar way to the `filebuf` class, but does not need to switch between the *get area* and *put area*, since no `std::streambuf` object can be created for both reading and writing. When the *get area* is empty and a read is done, the `underflow` virtual member function reads more characters and fills the *get area* again. When the *put area* is full and a write is done, the `overflow` virtual member function writes the characters and makes the *put area* empty again.

The `strstreambuf` class differs quite markedly from the `filebuf` and `std::streambuf` classes. Since there is no actual source or destination for the characters in `strstream` objects, the buffer itself takes on that role. When writing is occurring and the *put area* is full, the `overflow` virtual member function reallocates the buffer to a larger size (if possible), the *put area* is extended and the writing continues. If reading is occurring and the *get area* is empty, the `underflow` virtual member function checks to see if the *put area* is present and not empty. If so, the *get area* is extended to overlap the *put area*.

The *reserve area* is marked by two pointer values. The `base` member function returns the pointer to the start of the buffer. The `ebuf` member function returns the pointer to the end of the buffer (last character + 1). The `setb` protected member function is used to set both pointers.

Within the *reserve area*, the *get area* is marked by three pointer values. The `eback` member function returns a pointer to the start of the *get area*. The `egptr` member function returns a pointer to the end of the *get area* (last character + 1). The `gptr` member function returns the *get pointer*. The *get pointer* is a pointer to the next character to be extracted from the *get area*. Characters before the *get pointer* have already been consumed by the program, while characters at and after the *get pointer* have been read from their source and are buffered and waiting to be read by the program. The `setg` member function is used to set all three pointer values. If any of these pointers are `NULL`, there is no *get area*.

Also within the *reserve area*, the *put area* is marked by three pointer values. The `pbase` member function returns a pointer to the start of the *put area*. The `epptr` member function returns a pointer to the end of the *put area* (last character + 1). The `pptr` member function returns the *put pointer*. The *put pointer* is a pointer to the next available position into which a character may be stored. Characters before the *put pointer* are buffered and waiting to be written to their final destination, while character positions at and after the *put pointer* have yet to be written by the program. The `setp` member function is used to set all three pointer values. If any of these pointers are `NULL`, there is no *put area*.

Unbuffered I/O is also possible. If unbuffered, the `overflow` virtual member function is used to write single characters directly to their final destination without using the *put area*. Similarly, the `underflow` virtual member function is used to read single characters directly from their source without using the *get area*.

### Protected Member Functions

The following member functions are declared in the protected interface:

```
streambuf();
streambuf( char *, int );
virtual ~streambuf();
int allocate();
char *base() const;
char *eback() const;
int blen() const;
void setb( char *, char *, int );
char *eback() const;
char *gptra() const;
char *egptra() const;
void gbump( streamoff );
void setg( char *, char *, char *);
char *pbase() const;
char *pptra() const;
char *epptra() const;
void pbump( streamoff );
void setp( char *, char *);
int unbuffered( int );
int unbuffered() const;
virtual int doallocate();
```

### **Public Member Functions**

The following member functions are declared in the public interface:

```
int in_avail() const;
int out_waiting() const;
int snextc();
int sgetn( char *, int );
int speakc();
int sgetc();
int sgetchar();
int sbumpc();
void stoss();
int sputbackc( char );
int sputc( int );
int sputn( char const *, int );
void dbp();

virtual int do_sgetn( char *, int );
virtual int do_sputn( char const *, int );
virtual int pbackfail( int );
virtual int overflow( int = EOF ) = 0;
virtual int underflow() = 0;
```

## ***streambuf***

---

```
virtual streambuf *setbuf( char *, int );  
virtual streampos seekoff( streamoff, ios::seekdir,  
ios::openmode = ios::in|ios::out );  
virtual streampos seekpos( streampos,  
ios::openmode = ios::in|ios::out );  
virtual int sync();
```

**See Also:** filebuf, stdiobuf, strstreambuf



**Synopsis:**

```
#include <streambu.h>
protected:
int streambuf::allocate();
```

**Semantics:** The `allocate` protected member function works in tandem with the `doallocate` protected virtual member function to manage allocation of the `streambuf` object *reserve area*. Classes derived from the `streambuf` class should call the `allocate` protected member function, rather than the `doallocate` protected virtual member function. The `allocate` protected member function determines whether or not the `streambuf` object is allowed to allocate a buffer for use as the *reserve area*. If a *reserve area* already exists or if the `streambuf` object unbuffering state is non-zero, the `allocate` protected member function fails. Otherwise, it calls the `doallocate` protected virtual member function.

**Results:** The `allocate` protected member function returns `__NOT_EOF` on success, otherwise `EOF` is returned.

**See Also:** `streambuf::doallocate`, `underflow`, `overflow`

## ***streambuf::base()***

---

**Synopsis:**

```
#include <streambu.h>
protected:
char *streambuf::base() const;
```

**Semantics:** The base protected member function returns a pointer to the start of the *reserve area* that the `streambuf` object is using.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`      start of the *reserve area*.  
`ebuf()`      end of the *reserve area*.  
`blen()`      length of the *reserve area*.

`eback()`     start of the *get area*.  
`gptr()`      the *get pointer*.  
`egptr()`     end of the *get area*.

`pbase()`     start of the *put area*.  
`pptr()`      the *put pointer*.  
`epptr()`     end of the *put area*.

From `eback` to `gptr` are characters buffered and read. From `gptr` to `egptr` are characters buffered but not yet read. From `pbase` to `pptr` are characters buffered and not yet written. From `pptr` to `epptr` is unused buffer area.

**Results:** The base protected member function returns a pointer to the start of the *reserve area* that the `streambuf` object is using. If the `streambuf` object currently does not have a *reserve area*, `NULL` is returned.

**See Also:** `streambuf::blen`, `ebuf`, `setb`

**Synopsis:**

```
#include <streambu.h>
protected:
int streambuf::blen() const;
```

**Semantics:** The `blen` protected member function reports the length of the *reserve area* that the `streambuf` object is using.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`      start of the *reserve area*.  
`ebuf()`      end of the *reserve area*.  
`blen()`      length of the *reserve area*.

`eback()`     start of the *get area*.  
`gptr()`      the *get pointer*.  
`egptr()`     end of the *get area*.

`pbase()`     start of the *put area*.  
`pptr()`      the *put pointer*.  
`epptr()`     end of the *put area*.

From `eback` to `gptr` are characters buffered and read. From `gptr` to `egptr` are characters buffered but not yet read. From `pbase` to `pptr` are characters buffered and not yet written. From `pptr` to `epptr` is unused buffer area.

**Results:** The `blen` protected member function returns the length of the *reserve area* that the `streambuf` object is using. If the `streambuf` object currently does not have a *reserve area*, zero is returned.

**See Also:** `streambuf::base`, `ebuf`, `setb`

## ***streambuf::dbp()***

---

**Synopsis:**

```
#include <streambu.h>
public:
void streambuf::dbp();
```

**Semantics:** The `dbp` public member function dumps information about the `streambuf` object directly to `stdout`, and is used for debugging classes derived from the `streambuf` class.

The following is an example of what the `dbp` public member function dumps:

```
STREAMBUF Debug Info:
this   = 00030679, unbuffered = 0, delete_reserve = 1
base   = 00070010, ebuf = 00070094
eback  = 00000000, gptr = 00000000, egptr = 00000000
pbase  = 00070010, pptr = 00070010, epptr = 00070094
```

**Synopsis:**

```
#include <streambu.h>
public:
virtual int do_sgetn( char *buf, int len );
```

**Semantics:** The `do_sgetn` public virtual member function works in tandem with the `sgetn` member function to transfer *len* characters from the *get area* into *buf*.

Classes derived from the `streambuf` class should call the `sgetn` member function, rather than the `do_sgetn` public virtual member function.

**Derived Implementation Protocol:**

Classes derived from the `streambuf` class that implement the `do_sgetn` public virtual member function should support copying up to *len* characters from the source through the *get area* and into *buf*.

**Default Implementation:**

The default `do_sgetn` public virtual member function provided with the `streambuf` class calls the `underflow` virtual member function to fetch more characters and then copies the characters from the *get area* into *buf*.

**Results:** The `do_sgetn` public virtual member function returns the number of characters successfully transferred.

**See Also:** `streambuf::sgetn`

## ***streambuf::do\_sputn()***

---

**Synopsis:**

```
#include <streambu.h>
public:
virtual int do_sputn( char const *buf, int len );
```

**Semantics:** The `do_sputn` public virtual member function works in tandem with the `sputn` member function to transfer *len* characters from *buf* to the end of the *put area* and advances the *put pointer*.

Classes derived from the `streambuf` class should call the `sputn` member function, rather than the `do_sputn` public virtual member function.

### **Derived Implementation Protocol:**

Classes derived from the `streambuf` class that implement the `do_sputn` public virtual member function should support copying up to *len* characters from *buf* through the *put area* and out to the destination device.

### **Default Implementation:**

The default `do_sputn` public virtual member function provided with the `streambuf` class calls the `overflow` virtual member function to flush the *put area* and then copies the rest of the characters from *buf* into the *put area*.

**Results:** The `do_sputn` public virtual member function returns the number of characters successfully written. If an error occurs, this number may be less than *len*.

**See Also:** `streambuf::sputn`

**Synopsis:**

```
#include <streambu.h>
protected:
virtual int streambuf::doallocate();
```

**Semantics:** The `doallocate` protected virtual member function manages allocation of the `streambuf` object's *reserve area* in tandem with the `allocate` protected member function.

Classes derived from the `streambuf` class should call the `allocate` protected member function rather than the `doallocate` protected virtual member function.

The `doallocate` protected virtual member function does the actual memory allocation, and can be defined for each class derived from the `streambuf` class.

**Derived Implementation Protocol:**

Classes derived from the `streambuf` class should implement the `doallocate` protected virtual member function such that it does the following:

1. attempts to allocate an area of memory,
2. calls the `setb` protected member function to initialize the *reserve area* pointers,
3. performs any class specific operations required.

**Default Implementation:**

The default `doallocate` protected virtual member function provided with the `streambuf` class attempts to allocate a buffer area with the `operator new` intrinsic function. It then calls the `setb` protected member function to set up the pointers to the *reserve area*.

**Results:** The `doallocate` protected virtual member function returns `__NOT_EOF` on success, otherwise EOF is returned.

**See Also:** `streambuf::allocate`

## ***streambuf::eback()***

---

**Synopsis:**

```
#include <streambu.h>
protected:
char *streambuf::eback() const;
```

**Semantics:** The `eback` protected member function returns a pointer to the start of the *get area* within the *reserve area* used by the `streambuf` object.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`      start of the *reserve area*.  
`ebuf()`      end of the *reserve area*.  
`blen()`      length of the *reserve area*.

`eback()`      start of the *get area*.  
`gptra()`      the *get pointer*.  
`egptra()`      end of the *get area*.

`pbase()`      start of the *put area*.  
`pptra()`      the *put pointer*.  
`epptra()`      end of the *put area*.

From `eback` to `gptra` are characters buffered and read. From `gptra` to `egptra` are characters buffered but not yet read. From `pbase` to `pptra` are characters buffered and not yet written. From `pptra` to `epptra` is unused buffer area.

**Results:** The `eback` protected member function returns a pointer to the start of the *get area*. If the `streambuf` object currently does not have a *get area*, `NULL` is returned.

**See Also:** `streambuf::egptra`, `gptra`, `setg`



**Synopsis:**

```
#include <streambu.h>
protected:
char *streambuf::ebuf() const;
```

**Semantics:** The `ebuf` protected member function returns a pointer to the end of the *reserve area* that the `streambuf` object is using. The character pointed at is actually the first character past the end of the *reserve area*.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`      start of the *reserve area*.  
`ebuf()`      end of the *reserve area*.  
`blen()`      length of the *reserve area*.

`eback()`     start of the *get area*.  
`gptr()`      the *get pointer*.  
`egptr()`     end of the *get area*.

`pbase()`     start of the *put area*.  
`pptr()`      the *put pointer*.  
`epptr()`     end of the *put area*.

From `eback` to `gptr` are characters buffered and read. From `gptr` to `egptr` are characters buffered but not yet read. From `pbase` to `pptr` are characters buffered and not yet written. From `pptr` to `epptr` is unused buffer area.

**Results:** The `ebuf` protected member function returns a pointer to the end of the *reserve area*. If the `streambuf` object currently does not have a *reserve area*, `NULL` is returned.

**See Also:** `streambuf::base`, `blen`, `setb`

## ***streambuf::egptr()***

---

**Synopsis:** `#include <streambu.h>`  
`protected:`  
`char *streambuf::egptr() const;`

**Semantics:** The `egptr` protected member function returns a pointer to the end of the *get area* within the *reserve area* used by the `streambuf` object. The character pointed at is actually the first character past the end of the *get area*.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`      start of the *reserve area*.  
`ebuf()`      end of the *reserve area*.  
`blen()`      length of the *reserve area*.

`eback()`     start of the *get area*.  
`gptr()`      the *get pointer*.  
`egptr()`     end of the *get area*.

`pbase()`     start of the *put area*.  
`pptr()`     the *put pointer*.  
`epptr()`     end of the *put area*.

From `eback` to `gptr` are characters buffered and read. From `gptr` to `egptr` are characters buffered but not yet read. From `pbase` to `pptr` are characters buffered and not yet written. From `pptr` to `epptr` is unused buffer area.

**Results:** The `egptr` protected member function returns a pointer to the end of the *get area*. If the `streambuf` object currently does not have a *get area*, `NULL` is returned.

**See Also:** `streambuf::eback`, `gptr`, `setg`

**Synopsis:**

```
#include <streambu.h>
protected:
char *streambuf::epptr() const;
```

**Semantics:** The `epptr` protected member function returns a pointer to the end of the *put area* within the *reserve area* used by the `streambuf` object. The character pointed at is actually the first character past the end of the *put area*.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`      start of the *reserve area*.  
`ebuf()`      end of the *reserve area*.  
`blen()`      length of the *reserve area*.

`eback()`     start of the *get area*.  
`gptra()`     the *get pointer*.  
`egptra()`    end of the *get area*.

`pbase()`      start of the *put area*.  
`pptra()`      the *put pointer*.  
`epptra()`     end of the *put area*.

From `eback` to `gptra` are characters buffered and read. From `gptra` to `egptra` are characters buffered but not yet read. From `pbase` to `pptra` are characters buffered and not yet written. From `pptra` to `epptra` is unused buffer area.

**Results:** The `epptr` protected member function returns a pointer to the end of the *put area*. If the `streambuf` object currently does not have a *put area*, `NULL` is returned.

**See Also:** `streambuf::pbase`, `pptra`, `setp`

## ***streambuf::gbump()***

---

**Synopsis:** `#include <streambu.h>`  
`protected:`  
`void streambuf::gbump( streamoff offset );`

**Semantics:** The `gbump` protected member function increments the *get pointer* by the specified *offset*, without regard for the boundaries of the *get area*. The *offset* parameter may be positive or negative.

**Results:** The `gbump` protected member function returns nothing.

**See Also:** `streambuf::gptr`, `pbump`, `sbumpc`, `sputbackc`

**Synopsis:**

```
#include <streambu.h>
protected:
char *streambuf::gptr() const;
```

**Semantics:** The `gptr` protected member function returns a pointer to the next available character in the *get area* within the *reserve area* used by the `streambuf` object. This pointer is called the *get pointer*.

If the *get pointer* points beyond the end of the *get area*, all characters in the *get area* have been read by the program and a subsequent read causes the `underflow` virtual member function to be called to fetch more characters from the source to which the `streambuf` object is attached.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`      start of the *reserve area*.  
`ebuf()`      end of the *reserve area*.  
`blen()`      length of the *reserve area*.

`eback()`     start of the *get area*.  
`gptr()`      the *get pointer*.  
`egptr()`     end of the *get area*.

`pbase()`     start of the *put area*.  
`pptr()`     the *put pointer*.  
`epptr()`     end of the *put area*.

From `eback` to `gptr` are characters buffered and read. From `gptr` to `egptr` are characters buffered but not yet read. From `pbase` to `pptr` are characters buffered and not yet written. From `pptr` to `epptr` is unused buffer area.

**Results:** The `gptr` protected member function returns a pointer to the next available character in the *get area*. If the `streambuf` object currently does not have a *get area*, `NULL` is returned.

**See Also:** `streambuf::eback`, `egptr`, `setg`

## ***streambuf::in\_avail()***

---

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::in_avail() const;
```

**Semantics:** The `in_avail` public member function computes the number of input characters buffered in the *get area* that have not yet been read by the program. These characters can be read with a guarantee that no errors will occur.

**Results:** The `in_avail` public member function returns the number of buffered input characters.

**See Also:** `streambuf::egptr`, `gptr`

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::out_waiting() const;
```

**Semantics:** The `out_waiting` public member function computes the number of characters that have been buffered in the *put area* and not yet been written to the output device.

**Results:** The `out_waiting` public member function returns the number of buffered output characters.

**See Also:** `streambuf::pbase`, `pptr`

## ***streambuf::overflow()***

---

**Synopsis:**

```
#include <streambu.h>
public:
virtual int streambuf::overflow( int ch = EOF ) = 0;
```

**Semantics:** The `overflow` public virtual member function is used to flush the *put area* when it is full.

### **Derived Implementation Protocol:**

Classes derived from the `streambuf` class should implement the `overflow` public virtual member function so that it performs the following:

1. if no *reserve area* is present and the `streambuf` object is not unbuffered, allocate a *reserve area* using the `allocate` member function and set up the *reserve area* pointers using the `setb` protected member function,
2. flush any other uses of the *reserve area*,
3. write any characters in the *put area* to the `streambuf` object's destination,
4. set up the *put area* pointers to reflect the characters that were written,
5. return `__NOT_EOF` on success, otherwise return `EOF`.

### **Default Implementation:**

There is no default `streambuf` class implementation of the `overflow` public virtual member function. The `overflow` public virtual member function must be defined for all classes derived from the `streambuf` class.

**Results:** The `overflow` public virtual member function returns `__NOT_EOF` on success, otherwise `EOF` is returned.

**See Also:** `filebuf::overflow`, `stdiobuf::overflow`, `strstreambuf::overflow`



**Synopsis:**

```
#include <streambu.h>
public:
virtual int streambuf::pbackfail( int ch );
```

**Semantics:** The `pbackfail` public virtual member function is called by the `sputbackc` member function when the *get pointer* is at the beginning of the *get area*, and so there is no place to put the *ch* parameter.

**Derived Implementation Protocol:**

Classes derived from the `streambuf` class should implement the `pbackfail` public virtual member function such that it attempts to put *ch* back into the source of the stream.

**Default Implementation:**

The default `streambuf` class implementation of the `pbackfail` public virtual member function is to return EOF.

**Results:** If the `pbackfail` public virtual member function succeeds, it returns *ch*. Otherwise, EOF is returned.

## ***streambuf::pbase()***

---

**Synopsis:**

```
#include <streambu.h>
protected:
char *streambuf::pbase() const;
```

**Semantics:** The `pbase` protected member function returns a pointer to the start of the *put area* within the *reserve area* used by the `streambuf` object.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`        start of the *reserve area*.  
`ebuf()`        end of the *reserve area*.  
`blen()`        length of the *reserve area*.

`eback()`       start of the *get area*.  
`gptr()`        the *get pointer*.  
`egptr()`       end of the *get area*.

`pbase()`        start of the *put area*.  
`pptr()`        the *put pointer*.  
`epptr()`       end of the *put area*.

From `eback` to `gptr` are characters buffered and read. From `gptr` to `egptr` are characters buffered but not yet read. From `pbase` to `pptr` are characters buffered and not yet written. From `pptr` to `epptr` is unused buffer area.

**Results:** The `pbase` protected member function returns a pointer to the start of the *put area*. If the `streambuf` object currently does not have a *put area*, `NULL` is returned.

**See Also:** `streambuf::epptr`, `pptr`, `setp`

**Synopsis:** `#include <streambu.h>`  
`protected:`  
`void streambuf::pbump( streamoff offset );`

**Semantics:** The `pbump` protected member function increments the *put pointer* by the specified *offset*, without regard for the boundaries of the *put area*. The *offset* parameter may be positive or negative.

**Results:** The `pbump` protected member function returns nothing.

**See Also:** `streambuf::gbump`, `pbase`, `pptr`

## ***streambuf::pptr()***

---

**Synopsis:**

```
#include <streambu.h>
protected:
char *streambuf::pptr() const;
```

**Semantics:** The `pptr` protected member function returns a pointer to the next available space in the *put area* within the *reserve area* used by the `streambuf` object. This pointer is called the *put pointer*.

If the *put pointer* points beyond the end of the *put area*, the *put area* is full and a subsequent write causes the `overflow` virtual member function to be called to empty the *put area* to the device to which the `streambuf` object is attached.

The *reserve area*, *get area*, and *put area* pointer functions return the following values:

`base()`      start of the *reserve area*.  
`ebuf()`      end of the *reserve area*.  
`blen()`      length of the *reserve area*.

`eback()`     start of the *get area*.  
`gptr()`      the *get pointer*.  
`egptr()`     end of the *get area*.

`pbase()`     start of the *put area*.  
`pptr()`      the *put pointer*.  
`epptr()`     end of the *put area*.

From `eback` to `gptr` are characters buffered and read. From `gptr` to `egptr` are characters buffered but not yet read. From `pbase` to `pptr` are characters buffered and not yet written. From `pptr` to `epptr` is unused buffer area.

**Results:** The `pptr` protected member function returns a pointer to the next available space in the *put area*. If the `streambuf` object currently does not have a *put area*, `NULL` is returned.

**See Also:** `streambuf::epptr`, `pbase`, `setp`

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::sbumpc();
```

**Semantics:** The `sbumpc` public member function extracts the next available character from the *get area* and advances the *get pointer*. If no character is available, it calls the `underflow` virtual member function to fetch more characters from the source into the *get area*.

Due to the `sbumpc` member functions's awkward name, the `sgetchar` member function was added to take its place in the WATCOM implementation.

**Results:** The `sbumpc` public member function returns the next available character in the *get area*. If no character is available, EOF is returned.

**See Also:** `streambuf::gbump`, `sgetc`, `sgetchar`, `sgetn`, `snextc`, `sputbackc`

## ***streambuf::seekoff()***

---

**Synopsis:**

```
#include <streambu.h>
public:
virtual streampos streambuf::seekoff( streamoff offset,
ios::seekdir dir,
ios::openmode mode );
```

**Semantics:** The `seekoff` public virtual member function is used for positioning to a relative location within the `streambuf` object, and hence within the device that is connected to the `streambuf` object. The *offset* and *dir* parameters specify the relative change in position. The *mode* parameter controls whether the *get pointer* and/or the *put pointer* are repositioned.

### **Derived Implementation Protocol:**

Classes derived from the `streambuf` class should implement the `seekoff` virtual member function so that it uses its parameters in the following way.

The *mode* parameter may be `ios::in`, `ios::out`, or `ios::in|ios::out` and should be interpreted as follows, provided the interpretation is meaningful:

<code>ios::in</code>	the <i>get pointer</i> should be moved.
<code>ios::out</code>	the <i>put pointer</i> should be moved.
<code>ios::in ios::out</code>	both the <i>get pointer</i> and the <i>put pointer</i> should be moved.

If *mode* has any other value, the `seekoff` public virtual member function fails.

The *dir* parameter may be `ios::beg`, `ios::cur`, or `ios::end` and is interpreted in conjunction with the *offset* parameter as follows:

<code>ios::beg</code>	the <i>offset</i> is relative to the start and should be a positive value.
<code>ios::cur</code>	the <i>offset</i> is relative to the current position and may be positive (seek towards end) or negative (seek towards start).
<code>ios::end</code>	the <i>offset</i> is relative to the end and should be a negative value.

If the *dir* parameter has any other value, or the *offset* parameter does not have an appropriate sign, the `seekoff` public virtual member function fails.

### **Default Implementation:**

The default implementation of the `seekoff` public virtual member function provided by the `streambuf` class returns EOF.

**Results:** The `seekoff` public virtual member function returns the new position in the stream on success, otherwise EOF is returned.

**See Also:** `streambuf::seekpos`

## **836 Input/Output Classes**

**Synopsis:**

```
#include <streambu.h>
public:
virtual streampos streambuf::seekpos( streampos pos,
ios::openmode mode = ios::in|ios::out );
```

**Semantics:** The `seekpos` public virtual member function is used for positioning to an absolute location within the `streambuf` object, and hence within the device that is connected to the `streambuf` object. The `pos` parameter specifies the absolute position. The `mode` parameter controls whether the *get pointer* and/or the *put pointer* are repositioned.

**Derived Implementation Protocol:**

Classes derived from the `streambuf` class should implement the `seekpos` public virtual member function so that it uses its parameters in the following way.

The `mode` parameter may be `ios::in`, `ios::out`, or `ios::in|ios::out` and should be interpreted as follows, provided the interpretation is meaningful:

<code>ios::in</code>	the <i>get pointer</i> should be moved.
<code>ios::out</code>	the <i>put pointer</i> should be moved.
<code>ios::in ios::out</code>	both the <i>get pointer</i> and the <i>put pointer</i> should be moved.

If `mode` has any other value, the `seekpos` public virtual member function fails.

In general the `seekpos` public virtual member function is equivalent to calling the `seekoff` virtual member function with the offset set to `pos`, the direction set to `ios::beg` and the mode set to `mode`.

**Default Implementation:**

The default implementation of the `seekpos` public virtual member function provided by the `streambuf` class calls the `seekoff` virtual member function with the offset set to `pos`, the direction set to `ios::beg`, and the mode set to `mode`.

**Results:** The `seekpos` public virtual member function returns the new position in the stream on success, otherwise EOF is returned.

**See Also:** `streambuf::seekoff`

## ***streambuf::setb()***

---

**Synopsis:**

```
#include <streambu.h>
protected:
void streambuf::setb( char *base, char *ebuf, int autodel );
```

**Semantics:** The `setb` protected member function is used to set the pointers to the *reserve area* that the `streambuf` object is using.

The *base* parameter is a pointer to the start of the *reserve area* and corresponds to the value that the `base` member function returns.

The *ebuf* parameter is a pointer to the end of the *reserve area* and corresponds to the value that the `ebuf` member function returns.

The *autodel* parameter indicates whether or not the `streambuf` object can free the *reserve area* when the `streambuf` object is destroyed or when a new *reserve area* is set up in a subsequent call to the `setb` protected member function. If the *autodel* parameter is non-zero, the `streambuf` object can delete the *reserve area*, using the `operator delete` intrinsic function. Otherwise, a zero value indicates that the buffer will be deleted elsewhere.

If either of the *base* or *ebuf* parameters are `NULL` or if  $ebuf \leq base$ , the `streambuf` object does not have a buffer and input/output operations are unbuffered, unless another buffer is set up.

Note that the `setb` protected member function is used to set the *reserve area* pointers, while the `setbuf` protected member function is used to offer a buffer to the `streambuf` object.

**See Also:** `streambuf::base`, `blen`, `ebuf`, `setbuf`



**Synopsis:**

```
#include <streambu.h>
public:
virtual streambuf *streambuf::setbuf( char *buf, int len );
```

**Semantics:** The `setbuf` public virtual member function is used to offer a buffer specified by the *buf* and *len* parameters to the `streambuf` object for use as its *reserve area*. Note that the `setbuf` public virtual member function is used to offer a buffer, while the `setb` protected member function is used to set the *reserve area* pointers once a buffer has been accepted.

**Derived Implementation Protocol:**

Classes derived from the `streambuf` class may implement the `setbuf` public virtual member function if the default behavior is not suitable.

Derived classes that provide their own implementations of the `setbuf` public virtual member function may accept or reject the offered buffer. Often, if a buffer is already allocated, the offered buffer is rejected, as it may be difficult to transfer the information from the current buffer.

**Default Implementation:**

The default `setbuf` public virtual member function provided by the `streambuf` class rejects the buffer if one is already present.

If no buffer is present and either *buf* is `NULL` or *len* is zero, the offer is accepted and the `streambuf` object is unbuffered.

Otherwise, no buffer is present and one is specified. If *len* is less than five characters the buffer is too small and it is rejected. Otherwise, the buffer is accepted.

**Results:** The `setbuf` public virtual member function returns the address of the `streambuf` object if the offered buffer is accepted, otherwise `NULL` is returned.

**See Also:** `streambuf::setb`

## ***streambuf::setg()***

---

**Synopsis:** `#include <streambu.h>`  
`protected:`  
`void streambuf::setg( char *eback, char *gptr, char *egptr );`

**Semantics:** The `setg` protected member function is used to set the three *get area* pointers.

The *eback* parameter is a pointer to the start of the *get area* and corresponds to the value that the `eback` member function returns.

The *gptr* parameter is a pointer to the first available character in the *get area*, that is, the *get pointer*, and usually is greater than the *eback* parameter in order to accommodate a putback area. The *gptr* parameter corresponds to the value that the `gptr` member function returns.

The *egptr* parameter is a pointer to the end of the *get area* and corresponds to the value that the `egptr` member function returns.

If any of the three parameters are `NULL`, there is no *get area*.

**See Also:** `streambuf::eback`, `egptr`, `gptr`

**Synopsis:** `#include <streambu.h>`  
`protected:`  
`void streambuf::setp( char *pbase, char *eptr );`

**Semantics:** The `setp` protected member function is used to set the three *put area* pointers.

The *pbase* parameter is a pointer to the start of the *put area* and corresponds to the value that the `pbase` member function returns.

The *eptr* parameter is a pointer to the end of the *put area* and corresponds to the value that the `eptr` member function returns.

The *put pointer* is set to the *pbase* parameter value and corresponds to the value that the `pptr` member function returns.

If either parameter is `NULL`, there is no *put area*.

**See Also:** `streambuf::eptr`, `pbase`, `pptr`

## ***streambuf::sgetc()***

---

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::sgetc();
```

**Semantics:** The `sgetc` public member function returns the next available character in the *get area*. The *get pointer* is not advanced. If the *get area* is empty, the `underflow` virtual member function is called to fetch more characters from the source into the *get area*.

Due to the `sgetc` member function's confusing name (the C library `getc` function does advance the pointer), the `speekc` member function was added to take its place in the WATCOM implementation.

**Results:** The `sgetc` public member function returns the next available character in the *get area*. If no character is available, EOF is returned.

**See Also:** `streambuf::sbumpc`, `sgetchar`, `sgetn`, `snextc`, `speekc`

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::sgetchar();
```

**Semantics:** The `sgetchar` public member function extracts the next available character from the *get area* and advances the *get pointer*. If no character is available, it calls the `underflow` virtual member function to fetch more characters from the source into the *get area*.

Due to the `sbumpc` member functions's awkward name, the `sgetchar` member function was added to take its place in the WATCOM implementation.

**Results:** The `sgetchar` public member function returns the next available character in the *get area*. If no character is available, EOF is returned.

**See Also:** `streambuf::gbump`, `sgetc`, `sgetchar`, `sgetn`, `snextc`, `speekc`, `sputbackc`

## ***streambuf::sgetn()***

---

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::sgetn( char *buf, int len );
```

**Semantics:** The `sgetn` public member function transfers up to *len* characters from the *get area* into *buf*. If there are not enough characters in the *get area*, the `do_sgetn` virtual member function is called to fetch more.

Classes derived from the `streambuf` class should call the `sgetn` public member function, rather than the `do_sgetn` virtual member function.

**Results:** The `sgetn` public member function returns the number of characters transferred from the *get area* into *buf*.

**See Also:** `streambuf::do_sgetn`, `sbumpc`, `sgetc`, `sgetchar`, `speekc`

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::snextc();
```

**Semantics:** The `snextc` public member function advances the *get pointer* and then returns the character following the *get pointer*. The *get pointer* is left pointing at the returned character.

If the *get pointer* cannot be advanced, the `underflow` virtual member function is called to fetch more characters from the source into the *get area*.

**Results:** The `snextc` public member function advances the *get pointer* and returns the next available character in the *get area*. If there is no next available character, EOF is returned.

**See Also:** `streambuf::sbumpc`, `sgetc`, `sgetchar`, `sgetn`, `speekc`

## ***streambuf::speekc()***

---

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::speekc();
```

**Semantics:** The `speekc` public member function returns the next available character in the *get area*. The *get pointer* is not advanced. If the *get area* is empty, the `underflow` virtual member function is called to fetch more characters from the source into the *get area*.

Due to the `sgetc` member function's confusing name (the C library `getc` function does advance the pointer), the `speekc` member function was added to take its place in the WATCOM implementation.

**Results:** The `speekc` public member function returns the next available character in the *get area*. If no character is available, EOF is returned.

**See Also:** `streambuf::sbumpc`, `sgetc`, `sgetchar`, `sgetn`, `snextc`



**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::sputback( char ch );
```

**Semantics:** The `sputbackc` public member function is used to put a character back into the *get area*. The *ch* character specified must be the same as the character before the *get pointer*, otherwise the behavior is undefined. The *get pointer* is backed up by one position. At least four characters may be put back without any intervening reads.

**Results:** The `sputbackc` public member function returns *ch* on success, otherwise EOF is returned.

**See Also:** `streambuf::gbump`, `sbumpc`, `sgetchar`

## ***streambuf::putc()***

---

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::putc( int ch );
```

**Semantics:** The `putc` public member function adds the character *ch* to the end of the *put area* and advances the *put pointer*. If the *put area* is full before the character is added, the `overflow` virtual member function is called to empty the *put area* and write the character.

**Results:** The `putc` public member function returns *ch* on success, otherwise EOF is returned.

**See Also:** `streambuf::sgetc`, `sputn`

**Synopsis:**

```
#include <streambu.h>
public:
int streambuf::sputn( char const *buf, int len );
```

**Semantics:** The `sputn` public member function transfers up to *len* characters from *buf* to the end of the *put area* and advance the *put pointer*. If the *put area* is full or becomes full and more characters are to be written, the `do_sputn` virtual member function is called to empty the *put area* and finish writing the characters.

Classes derived from the `streambuf` class should call the `sputn` public member function, rather than the `do_sputn` virtual member function.

**Results:** The `sputn` public member function returns the number of characters successfully written. If an error occurs, this number may be less than *len*.

**See Also:** `streambuf::do_sputn`, `sputc`

## ***streambuf::stoss()***

---

**Synopsis:** `#include <streambu.h>`  
`public:`  
`void streambuf::stoss();`

**Semantics:** The `stoss` public member function advances the *get pointer* by one without returning a character. If the *get area* is empty, the `underflow` virtual member function is called to fetch more characters and then the *get pointer* is advanced.

**See Also:** `streambuf::gbump`, `sbumpc`, `sgetchar`, `snextc`

**Synopsis:**

```
#include <streambu.h>
protected:
streambuf::streambuf();
```

**Semantics:** This form of the protected `streambuf` constructor creates an empty `streambuf` object with all fields initialized to zero. No *reserve area* is yet allocated, but the `streambuf` object is buffered unless a subsequent call to the `setbuf` or `unbuffered` member functions dictate otherwise.

**Results:** This form of the protected `streambuf` constructor creates an initialized `streambuf` object with no associated *reserve area*.

**See Also:** `~streambuf`

## ***streambuf::streambuf()***

---

**Synopsis:** `#include <streambu.h>`  
`protected:`  
`streambuf::streambuf( char *buf, int len );`

**Semantics:** This form of the protected `streambuf` constructor creates an empty `streambuf` object with all fields initialized to zero. The *buf* and *len* parameters are passed to the `setbuf` member function, which sets up the buffer (if specified), or makes the `streambuf` object unbuffered (if the *buf* parameter is `NULL` or the *len* parameter is not positive).

**Results:** This form of the protected `streambuf` constructor creates an initialized `streambuf` object with an associated *reserve area*.

**See Also:** `~streambuf`, `setbuf`

**Synopsis:** `#include <streambu.h>`  
`protected:`  
`virtual streambuf::~~streambuf();`

**Semantics:** The `streambuf` object is destroyed. If the buffer was allocated by the `streambuf` object, it is freed. Otherwise, the buffer is not freed and must be freed by the user of the `streambuf` object. The call to the protected `~streambuf` destructor is inserted implicitly by the compiler at the point where the `streambuf` object goes out of scope.

**Results:** The `streambuf` object is destroyed.

**See Also:** `streambuf`

## ***streambuf::sync()***

---

**Synopsis:**

```
#include <streambu.h>
public:
virtual int streambuf::sync();
```

**Semantics:** The `sync` public virtual member function is used to synchronize the `streambuf` object's *get area* and *put area* with the associated device.

**Derived Implementation Protocol:**

Classes derived from the `streambuf` class should implement the `sync` public virtual member function such that it attempts to perform the following:

1. flush the *put area*,
2. discard the contents of the *get area* and reposition the stream device so that the discarded characters may be read again.

**Default Implementation:**

The default implementation of the `sync` public virtual member function provided by the `streambuf` class takes no action. It succeeds if the *get area* and the *put area* are empty, otherwise it fails.

**Results:** The `sync` public virtual member function returns `__NOT_EOF` on success, otherwise `EOF` is returned.



**Synopsis:**

```
#include <streambu.h>
protected:
int ios::unbuffered() const;
int ios::unbuffered( int unbuf );
```

**Semantics:** The `unbuffered` protected member function is used to query and/or set the unbuffering state of the `streambuf` object. A non-zero unbuffered state indicates that the `streambuf` object is unbuffered. An unbuffered state of zero indicates that the `streambuf` object is buffered.

The first form of the `unbuffered` protected member function is used to query the current unbuffering state.

The second form of the `unbuffered` protected member function is used to set the unbuffering state to *unbuf*.

Note that the unbuffering state only affects the `allocate` protected member function, which does nothing if the unbuffering state is non-zero. Setting the unbuffering state to a non-zero value does not mean that future I/O operations will be unbuffered.

To determine if current I/O operations are unbuffered, use the `base` protected member function. A return value of `NULL` from the `base` protected member function indicates that unbuffered I/O operations will be used.

**Results:** The `unbuffered` protected member function returns the previous unbuffered state.

**See Also:** `streambuf::allocate`, `pbase`, `setbuf`

## ***streambuf::underflow()***

---

**Synopsis:**

```
#include <streambu.h>
public:
virtual int streambuf::underflow() = 0;
```

**Semantics:** The `underflow` public virtual member function is used to fill the *get area* when it is empty.

### **Derived Implementation Protocol:**

Classes derived from the `streambuf` class should implement the `underflow` public virtual member function so that it performs the following:

1. if no *reserve area* is present and the `streambuf` object is buffered, allocate the *reserve area* using the `allocate` member function and set up the *reserve area* pointers using the `setb` protected member function,
2. flush any other uses of the *reserve area*,
3. read some characters from the `streambuf` object's source into the *get area*,
4. set up the *get area* pointers to reflect the characters that were read,
5. return the first character of the *get area*, or EOF if no characters could be read.

### **Default Implementation:**

There is no default `streambuf` class implementation of the `underflow` public virtual member function. The `underflow` public virtual member function must be defined for all classes derived from the `streambuf` class.

**Results:** The `underflow` public virtual member function returns the first character read into the *get area*, or EOF if no characters could be read.

**See Also:** `filebuf::underflow`, `stdiobuf::underflow`, `strstreambuf::underflow`

**Declared:** `strstream.h`

**Derived from:**

`stringstreambase`, `iostream`

The `strstream` class is used to create and write to string stream objects.

The `strstream` class provides little of its own functionality. Derived from the `stringstreambase` and `iostream` classes, its constructors and destructor provide simplified access to the appropriate equivalents in those base classes. The member functions provide specialized access to the string stream object.

Of the available I/O stream classes, creating a `strstream` object is the preferred method of performing read and write operations on a string stream.

**Public Member Functions**

The following member functions are declared in the public interface:

```
strstream();
strstream( char *,
int,
ios::openmode = ios::in|ios::out );
strstream( signed char *,
int,
ios::openmode = ios::in|ios::out );
strstream( unsigned char *,
int,
ios::openmode = ios::in|ios::out );
~strstream();
char *str();
```

**See Also:** `istrstream`, `ostrstream`, `stringstreambase`

## ***stringstream::str()***

---

**Synopsis:**

```
#include <strstream.h>
public:
char *stringstream::str();
```

**Semantics:** The `str` public member function creates a pointer to the buffer being used by the `stringstream` object. If the `stringstream` object was created without dynamic allocation (static mode), the pointer is the same as the buffer pointer passed in the constructor.

For `stringstream` objects using dynamic allocation, the `str` public member function makes an implicit call to the `strstreambuf::freeze` member function. If nothing has been written to the `stringstream` object, the returned pointer will be `NULL`.

Note that the buffer does not necessarily end with a null character. If the pointer returned by the `str` public member function is to be interpreted as a C string, it is the program's responsibility to ensure that the null character is present.

**Results:** The `str` public member function returns a pointer to the buffer being used by the `stringstream` object.

**See Also:** `strstreambuf::str`, `strstreambuf::freeze`

**Synopsis:**

```
#include <strstrea.h>
public:
strstream::strstream();
```

**Semantics:** This form of the public `strstream` constructor creates an empty `strstream` object. Dynamic allocation is used. The inherited stream member functions can be used to access the `strstream` object. Note that the *get pointer* and *put pointer* are not necessarily pointing at the same location, so moving one pointer (e.g. by doing a write) does not affect the location of the other pointer.

**Results:** This form of the public `strstream` constructor creates an initialized, empty `strstream` object.

**See Also:** `~strstream`

## ***strstream::strstream()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
strstream::strstream( char *str,
int len,
ios::openmode mode );
strstream::strstream( signed char *str,
int len,
ios::openmode mode );
strstream::strstream( unsigned char *str,
int len,
ios::openmode mode );
```

**Semantics:** These forms of the public `strstream` constructor create an initialized `strstream` object. Dynamic allocation is not used. The buffer is specified by the `str` and `len` parameters. If the `ios::append` or `ios::atend` bits are set in the `mode` parameter, the `str` parameter is assumed to contain a C string terminated by a null character, and writing commences at the null character. Otherwise, writing commences at `str`. Reading commences at `str`.

**Results:** This form of the public `strstream` constructor creates an initialized `strstream` object.

**See Also:** `~strstream`

**Synopsis:** `#include <strstrea.h>`  
`public:`  
`strstream::~~strstream();`

**Semantics:** The public `~strstream` destructor does not do anything explicit. The call to the public `~strstream` destructor is inserted implicitly by the compiler at the point where the `strstream` object goes out of scope.

**Results:** The `strstream` object is destroyed.

**See Also:** `strstream`

## ***strstreambase***

---

**Declared:** `strstrea.h`

**Derived from:**  
`ios`

**Derived by:** `istrstream`, `ostrstream`, `strstream`

The `strstreambase` class is a base class that provides common functionality for the three string stream-based classes, `istrstream`, `ostrstream` and `strstream`. The `strstreambase` class is derived from the `ios` class which provides the stream state information. The `strstreambase` class provides constructors for string stream objects and one member function.

### **Protected Member Functions**

The following member functions are declared in the protected interface:

```
strstreambase();  
strstreambase( char *, int, char * = 0 );  
~strstreambase();
```

### **Public Member Functions**

The following member function is declared in the public interface:

```
strstreambuf *rdbuf() const;
```

**See Also:** `istrstream`, `ostrstream`, `strstream`, `strstreambuf`



**Synopsis:** `#include <strstrea.h>`  
`public:`  
`strstreambuf *strstreambase::rdbuf() const;`

**Semantics:** The `rdbuf` public member function creates a pointer to the `strstreambuf` associated with the `strstreambase` object. Since the `strstreambuf` object is embedded within the `strstreambase` object, this function never returns `NULL`.

**Results:** The `rdbuf` public member function returns a pointer to the `strstreambuf` associated with the `strstreambase` object.

## ***strstreambase::strstreambase()***

---

**Synopsis:** `#include <strstrea.h>`  
`protected:`  
`strstreambase::strstreambase();`

**Semantics:** This form of the protected `strstreambase` constructor creates a `strstreambase` object that is initialized, but empty. Dynamic allocation is used to store characters. No buffer is allocated. A buffer is be allocated when data is first written to the `strstreambase` object.

This form of the protected `strstreambase` constructor is only used implicitly by the compiler when it generates a constructor for a derived class.

**Results:** The protected `strstreambase` constructor creates an initialized `strstreambase` object.

**See Also:** `~strstreambase`

**Synopsis:**

```
#include <strstrea.h>
protected:
strstreambase::strstreambase( char *str,
int len,
char *pstart );
```

**Semantics:** This form of the protected `strstreambase` constructor creates a `strstreambase` object that is initialized and uses the buffer specified by the *str* and *len* parameters as its *reserve area* within the associated `strstreambuf` object. Dynamic allocation is not used.

This form of the protected `strstreambase` constructor is unlikely to be explicitly used, except in the member initializer list for the constructor of a derived class.

The *str*, *len* and *pstart* parameters are interpreted as follows:

1. The buffer starts at *str*.
2. If *len* is positive, the buffer is *len* characters long.
3. If *len* is zero, *str* is a pointer to a C string which is terminated by a null character, and the length of the buffer is the length of the string.
4. If *len* is negative, the buffer is unbounded. This last form should be used with extreme caution, since no buffer is truly unlimited in size and it would be easy to write beyond the available space.
5. If the *pstart* parameter is `NULL`, the `strstreambase` object is read-only.
6. Otherwise, *pstart* divides the buffer into two regions. The *get area* starts at *str* and ends at *pstart*-1. The *put area* starts at *pstart* and goes to the end of the buffer.

**Results:** The protected `strstreambase` constructor creates an initialized `strstreambase` object.

**See Also:** `~strstreambase`

## ***strstreambase::~~strstreambase()***

---

**Synopsis:** `#include <strstrea.h>`  
`protected:`  
`strstreambase::~~strstreambase( ) ;`

**Semantics:** The protected `~strstreambase` destructor does not do anything explicit. The call to the protected `~strstreambase` destructor is inserted implicitly by the compiler at the point where the `strstreambase` object goes out of scope.

**Results:** The `strstreambase` object is destroyed.

**See Also:** `strstreambase`

**Declared:** `strstrea.h`

**Derived from:**

`streambuf`

The `strstreambuf` class is derived from the `streambuf` class and provides additional functionality required to write characters to and read characters from a string buffer. Read and write operations can occur at different positions in the string buffer, since the *get pointer* and *put pointer* are not necessarily connected. Seek operations are also supported.

The *reserve area* used by the `strstreambuf` object may be either fixed in size or dynamic. Generally, input strings are of a fixed size, while output streams are dynamic, since the final size may not be predictable. For dynamic buffers, the `strstreambuf` object automatically grows the buffer when necessary.

The `strstreambuf` class differs quite markedly from the `filebuf` and `stdiobuf` classes. Since there is no actual source or destination for the characters in `strstream` objects, the buffer itself takes on that role. When writing is occurring and the *put area* is full, the `overflow` virtual member function reallocates the buffer to a larger size (if possible), the *put area* is extended and the writing continues. If reading is occurring and the *get area* is empty, the `underflow` virtual member function checks to see if the *put area* is present and not empty. If so, the *get area* is extended to overlap the *put area*.

C++ programmers who wish to use string streams without deriving new objects will probably never explicitly create or use a `strstreambuf` object.

**Protected Member Functions**

The following member function is declared in the protected interface:

```
virtual int doallocate();
```

**Public Member Functions**

The following member functions are declared in the public interface:

```
strstreambuf();  
strstreambuf( int );  
strstreambuf( void *(*)( long ), void (*)( void * ) );  
strstreambuf( char *, int, char * = 0 );  
~strstreambuf();  
int alloc_size_increment( int );  
void freeze( int = 1 );  
char *str();  
virtual int overflow( int = EOF );
```

## ***strstreambuf***

---

```
virtual int underflow();
virtual streambuf *setbuf( char *, int );
virtual streampos seekoff( streamoff,
ios::seekdir,
ios::openmode );
virtual int sync();
```

**See Also:** streambuf, strstreambase

**Synopsis:**

```
#include <strstrea.h>
public:
int strstreambuf::alloc_size_increment( int increment );
```

**Semantics:** The `alloc_size_increment` public member function modifies the allocation size used when the buffer is first allocated or reallocated by dynamic allocation. The *increment* parameter is added to the previous allocation size for future use.

This function is a WATCOM extension.

**Results:** The `alloc_size_increment` public member function returns the previous value of the allocation size.

**See Also:** `strstreambuf::doallocate`, `setbuf`

## ***strstreambuf::doallocate()***

---

**Synopsis:**

```
#include <strstrea.h>
protected:
virtual int strstreambuf::doallocate();
```

**Semantics:** The `doallocate` protected virtual member function is called by the `allocate` member function when it is determined that the *put area* is full and needs to be extended.

The `doallocate` protected virtual member function performs the following steps:

1. If dynamic allocation is not being used, the `doallocate` protected virtual member function fails.
2. A new size for the buffer is determined. If the allocation size is bigger than the current size, the allocation size is used. Otherwise, the buffer size is increased by `DEFAULT_MAINBUF_SIZE`, which is 512.
3. A new buffer is allocated. If an allocation function was specified in the constructor for the `strstreambuf` object, that allocation function is used, otherwise the `operator new` intrinsic function is used. If the allocation fails, the `doallocate` protected virtual member function fails.
4. If necessary, the contents of the *get area* are copied to the newly allocated buffer and the *get area* pointers are adjusted accordingly.
5. The contents of the *put area* are copied to the newly allocated buffer and the *put area* pointers are adjusted accordingly, extending the *put area* to the end of the new buffer.
6. The old buffer is freed. If a free function was specified in the constructor for the `strstreambuf` object, that free function is used, otherwise the `operator delete` intrinsic function is used.

**Results:** The `doallocate` protected virtual member function returns `__NOT_EOF` on success, otherwise `EOF` is returned.

**See Also:** `strstreambuf::alloc_size_increment`, `setbuf`



**Synopsis:**

```
#include <strstrea.h>
public:
void strstreambuf::freeze( int frozen = 1 );
```

**Semantics:** The `freeze` public member function enables and disables automatic deletion of the *reserve area*. If the `freeze` public member function is called with no parameter or a non-zero parameter, the `strstreambuf` object is frozen. If the `freeze` public member function is called with a zero parameter, the `strstreambuf` object is unfrozen.

A frozen `strstreambuf` object does not free the *reserve area* in the destructor. If the `strstreambuf` object is destroyed while it is frozen, it is the program's responsibility to also free the *reserve area*.

If characters are written to the `strstreambuf` object while it is frozen, the effect is undefined since the *reserve area* may be reallocated and therefore may move. However, if the `strstreambuf` object is frozen and then unfrozen, characters may be written to it.

**Results:** The `freeze` public member function returns the previous frozen state.

**See Also:** `strstreambuf::str`, `~strstreambuf`

## ***strstreambuf::overflow()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
virtual int strstreambuf::overflow( int ch = EOF );
```

**Semantics:** The `overflow` public virtual member function provides the output communication between the `streambuf` member functions and the `strstreambuf` object. Member functions in the `streambuf` class call the `overflow` public virtual member function when the *put area* is full. The `overflow` public virtual member function attempts to grow the *put area* so that writing may continue.

The `overflow` public virtual member function performs the following steps:

1. If dynamic allocation is not being used, the *put area* cannot be extended, so the `overflow` public virtual member function fails.
2. If dynamic allocation is being used, a new buffer is allocated using the `doallocate` member function. It handles copying the contents of the old buffer to the new buffer and discarding the old buffer.
3. If the *ch* parameter is not `EOF`, it is added to the end of the extended *put area* and the *put pointer* is advanced.

**Results:** The `overflow` public virtual member function returns `__NOT_EOF` when it successfully extends the *put area*, otherwise `EOF` is returned.

**See Also:** `streambuf::overflow`  
`strstreambuf::underflow`

**Synopsis:**

```
#include <strstrea.h>
public:
virtual streampos strstreambuf::seekoff( streamoff offset,
ios::seekdir dir,
ios::openmode mode );
```

**Semantics:** The `seekoff` public virtual member function positions the *get pointer* and/or *put pointer* to the specified position in the *reserve area*. If the *get pointer* is moved, it is moved to a position relative to the start of the *reserve area* (which is also the start of the *get area*). If a position is specified that is beyond the end of the *get area* but is in the *put area*, the *get area* is extended to include the *put area*. If the *put pointer* is moved, it is moved to a position relative to the start of the *put area*, **not** relative to the start of the *reserve area*.

The `seekoff` public virtual member function seeks *offset* bytes from the position specified by the *dir* parameter.

The *mode* parameter may be `ios::in`, `ios::out`, or `ios::in|ios::out` and should be interpreted as follows, provided the interpretation is meaningful:

<code>ios::in</code>	the <i>get pointer</i> should be moved.
<code>ios::out</code>	the <i>put pointer</i> should be moved.
<code>ios::in ios::out</code>	both the <i>get pointer</i> and the <i>put pointer</i> should be moved.

If *mode* has any other value, the `seekoff` public virtual member function fails. `ios::in|ios::out` is not valid if the *dir* parameter is `ios::cur`.

The *dir* parameter may be `ios::beg`, `ios::cur`, or `ios::end` and is interpreted in conjunction with the *offset* parameter as follows:

<code>ios::beg</code>	the <i>offset</i> is relative to the start and should be a positive value.
<code>ios::cur</code>	the <i>offset</i> is relative to the current position and may be positive (seek towards end) or negative (seek towards start).
<code>ios::end</code>	the <i>offset</i> is relative to the end and should be a negative value.

If the *dir* parameter has any other value, or the *offset* parameter does not have an appropriate sign, the `seekoff` public virtual member function fails.

**Results:** The `seekoff` public virtual member function returns the new position in the file on success, otherwise EOF is returned. If both or `ios::in|ios::out` are specified and the *dir* parameter is `ios::cur` the returned position refers to the *put pointer*.

## ***strstreambuf::setbuf()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
virtual streambuf *strstreambuf::setbuf( char *, int size );
```

**Semantics:** The `setbuf` public virtual member function is used to control the size of the allocations when the `strstreambuf` object is using dynamic allocation. The first parameter is ignored. The next time an allocation is required, at least the number of characters specified in the `size` parameter is allocated. If the specified size is not sufficient, the allocation reverts to its default behavior, which is to extend the buffer by `DEFAULT_MAINBUF_SIZE`, which is 512 characters.

If a program is going to write a large number of characters to the `strstreambuf` object, it should call the `setbuf` public virtual member function to indicate the size of the next allocation, to prevent multiple allocations as the buffer gets larger.

**Results:** The `setbuf` public virtual member function returns a pointer to the `strstreambuf` object.

**See Also:** `strstreambuf::alloc_size_increment`, `doallocate`

**Synopsis:**

```
#include <strstrea.h>
public:
char *strstreambuf::str();
```

**Semantics:** The `str` public member function freezes the `strstreambuf` object and returns a pointer to the *reserve area*. This pointer remains valid after the `strstreambuf` object is destroyed provided the `strstreambuf` object remains frozen, since the destructor does not free the *reserve area* if it is frozen.

The returned pointer may be `NULL` if the `strstreambuf` object is using dynamic allocation but has not yet had anything written to it.

If the `strstreambuf` object is not using dynamic allocation, the pointer returned by the `str` public member function is the same buffer pointer provided to the constructor. For a `strstreambuf` object using dynamic allocation, the pointer points to a dynamically allocated area.

Note that the *reserve area* does not necessarily end with a null character. If the pointer returned by the `str` public member function is to be interpreted as a C string, it is the program's responsibility to ensure that the null character is present.

**Results:** The `str` public member function returns a pointer to the *reserve area* and freezes the `strstreambuf` object.

**See Also:** `strstreambuf::freeze`

## ***strstreambuf::strstreambuf()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
strstreambuf::strstreambuf();
```

**Semantics:** This form of the public `strstreambuf` constructor creates an empty `strstreambuf` object that uses dynamic allocation. No *reserve area* is allocated to start. Whenever characters are written to extend the `strstreambuf` object, the *reserve area* is reallocated and copied as required. The size of allocation is determined by the `strstreambuf` object unless the `setbuf` or `alloc_size_increment` member functions are called to change the allocation size. The default allocation size is determined by the constant `DEFAULT_MAINBUF_SIZE`, which is 512.

**Results:** This form of the public `strstreambuf` constructor creates a `strstreambuf` object.

**See Also:** `strstreambuf::doallocate`, `~strstreambuf`

**Synopsis:**

```
#include <strstrea.h>
public:
strstreambuf::strstreambuf( int alloc_size );
```

**Semantics:** This form of the public `strstreambuf` constructor creates an empty `strstreambuf` object that uses dynamic allocation. No buffer is allocated to start. Whenever characters are written to extend the `strstreambuf` object, the *reserve area* is reallocated and copied as required. The size of the first allocation is determined by the *alloc\_size* parameter, unless changed by a call to the `setbuf` or `alloc_size_increment` member functions.

Note that the *alloc\_size* parameter is the starting *reserve area* size. When the *reserve area* is reallocated, the `strstreambuf` object uses `DEFAULT_MAINBUF_SIZE` to increase the *reserve area* size, unless the `setbuf` or `alloc_size_increment` member functions have been called to specify a new allocation size.

**Results:** This form of the public `strstreambuf` constructor creates a `strstreambuf` object.

**See Also:** `strstreambuf::alloc_size_increment`, `doallocate`, `setbuf`, `~strstreambuf`

## ***strstreambuf::strstreambuf()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
strstreambuf::strstreambuf( void * (*alloc_fn)( long ),
void (*free_fn)( void * ) );
```

**Semantics:** This form of the public `strstreambuf` constructor creates an empty `strstreambuf` object that uses dynamic allocation. No buffer is allocated to start. Whenever characters are written to extend the `strstreambuf` object, the *reserve area* is reallocated and copied as required, using the specified *alloc\_fn* and *free\_fn* functions. The size of allocation is determined by the class unless the `setbuf` or `alloc_size_increment` member functions are called to change the allocation size. The default allocation size is determined by the constant `DEFAULT_MAINBUF_SIZE`, which is 512.

When a new *reserve area* is allocated, the function specified by the *alloc\_fn* parameter is called with a `long` integer value indicating the number of bytes to allocate. If *alloc\_fn* is `NULL`, the operator `new` intrinsic function is used. Likewise, when the *reserve area* is freed, the function specified by the *free\_fn* parameter is called with the pointer returned by the *alloc\_fn* function as the parameter. If *free\_fn* is `NULL`, the operator `delete` intrinsic function is used.

**Results:** This form of the public `strstreambuf` constructor creates a `strstreambuf` object.

**See Also:** `strstreambuf::alloc_size_increment`, `doallocate`, `setbuf`, `~strstreambuf`



**Synopsis:**

```
#include <strstrea.h>
public:
strstreambuf::strstreambuf( char *str,
int len,
char *pstart = NULL );
strstreambuf::strstreambuf( signed char *str,
int len,
signed char *pstart = NULL );
strstreambuf::strstreambuf( unsigned char *str,
int len,
unsigned char *pstart = NULL );
```

**Semantics:** This form of the public `strstreambuf` constructor creates a `strstreambuf` object that does not use dynamic allocation (unless `str` is `NULL`). The `strstreambuf` object is said to be using static allocation. The `str` and `len` parameters specify the bounds of the *reserve area*.

The `str`, `len` and `pstart` parameters are interpreted as follows:

1. The buffer starts at `str`.
2. If `len` is positive, the buffer is `len` characters long.
3. If `len` is zero, `str` is a pointer to a C string which is terminated by a null character, and the length of the buffer is the length of the string.
4. If `len` is negative, the buffer is unbounded. This last form should be used with extreme caution, since no buffer is truly unlimited in size and it would be easy to write beyond the available space.
5. If the `pstart` parameter is `NULL`, the `strstreambuf` object is read-only.
6. Otherwise, `pstart` divides the buffer into two regions. The *get area* starts at `str` and ends at `pstart-1`. The *put area* starts at `pstart` and goes to the end of the buffer.

If the *get area* is exhausted and characters have been written to the *put area*, the *get area* is extended to include the *put area*.

The *get pointer* and *put pointer* do not necessarily point at the same position in the *reserve area*, so a read followed by a write does not imply that the write stores following the last character read. The *get pointer* is positioned following the last read operation, and the *put pointer* is positioned following the last write operation, unless the `seekoff` member function has been used to reposition the pointer(s).

Note that if `str` is `NULL` the effect is to create an empty dynamic `strstreambuf` object.

## ***strstreambuf::strstreambuf()***

---

**Results:** This form of the public `strstreambuf` constructor creates a `strstreambuf` object.

**See Also:** `~strstreambuf`

**Synopsis:**

```
#include <strstrea.h>
public:
strstreambuf::~~strstreambuf();
```

**Semantics:** The public `~strstreambuf` destructor destroys the `strstreambuf` object after discarding the *reserve area*. The *reserve area* is discarded only if the `strstreambuf` object is using dynamic allocation and is not frozen. The *reserve area* is freed using the `free` function specified by the form of the constructor that allows specification of the `allocate` and `free` functions, or using the `operator delete` intrinsic function. If the `strstreambuf` object is frozen or using static allocation, the user of the `strstreambuf` object must have a pointer to the *reserve area* and is responsible for freeing it. The call to the public `~strstreambuf` destructor is inserted implicitly by the compiler at the point where the `strstreambuf` object goes out of scope.

**Results:** The `strstreambuf` object is destroyed.

**See Also:** `strstreambuf`

## ***strstreambuf::sync()***

---

**Synopsis:**

```
#include <strstrea.h>
public:
virtual int strstreambuf::sync();
```

**Semantics:** The `sync` public virtual member function does nothing because there is no external device with which to synchronize.

**Results:** The `sync` public virtual member function returns `__NOT_EOF`.

**Synopsis:**

```
#include <strstrea.h>
public:
virtual int strstreambuf::underflow();
```

**Semantics:** The `underflow` public virtual member function provides the input communication between the `streambuf` member functions and the `strstreambuf` object. Member functions in the `streambuf` class call the `underflow` public virtual member function when the *get area* is empty.

If there is a non-empty *put area* present following the *get area*, the *get area* is extended to include the *put area*, allowing the input operation to continue using the *put area*. Otherwise the *get area* cannot be extended.

**Results:** The `underflow` public virtual member function returns the first available character in the *get area* on successful extension, otherwise EOF is returned.

**See Also:** `streambuf::underflow`  
`strstreambuf::overflow`



---

# 19 String Class

This class is used to store arbitrarily long sequences of characters in memory. Objects of this type may be concatenated, substringed, compared and searched without the need for memory management by the user. Unlike a C string, this object has no delimiting character, so any character in the collating sequence, or character set, may be stored in an object.

The class documented here is the Open Watcom legacy string class. It is not related to the `std::basic_string` class template nor to its corresponding specialization `std::string`.

**Declared:** string.hpp

The `String` class is used to store arbitrarily long sequences of characters in memory. Objects of this type may be concatenated, substringed, compared and searched without the need for memory management by the user. Unlike a C string, a `String` object has no delimiting character, so any character in the collating sequence, or character set, may be stored in a `String` object.

### Public Functions

The following constructors and destructors are declared:

```
String();
String( size_t, capacity );
String( String const &, size_t = 0, size_t = NPOS );
String( char const *, size_t = NPOS );
String( char, size_t = 1 );
~String();
```

The following member functions are declared:

```
operator char const *();
operator char() const;
String &operator =( String const & );
String &operator =( char const * );
String &operator +=( String const & );
String &operator +=( char const * );
String operator ()( size_t, size_t ) const;
char &operator ()( size_t );
char const &operator []( size_t ) const;
char &operator []( size_t );
int operator !() const;
size_t length() const;
char const &get_at( size_t ) const;
void put_at( size_t, char );
int match( String const & ) const;
int match( char const * ) const;
int index( String const &, size_t = 0 ) const;
int index( char const *, size_t = 0 ) const;
String upper() const;
String lower() const;
int valid() const;
int alloc_mult_size() const;
int alloc_mult_size( int );
```

The following friend functions are declared:



```
friend int operator ==( String const &, String const & );
friend int operator ==( String const &, char const * );
friend int operator ==( char const *, String const & );
friend int operator ==( String const &, char );
friend int operator ==( char, String const & );
friend int operator !=( String const &, String const & );
friend int operator !=( String const &, char const * );
friend int operator !=( char const *, String const & );
friend int operator !=( String const &, char );
friend int operator !=( char, String const & );
friend int operator <( String const &, String const & );
friend int operator <( String const &, char const * );
friend int operator <( char const *, String const & );
friend int operator <( String const &, char );
friend int operator <( char, String const & );
friend int operator <=( String const &, String const & );
friend int operator <=( String const &, char const * );
friend int operator <=( char const *, String const & );
friend int operator <=( String const &, char );
friend int operator <=( char, String const & );
friend int operator >( String const &, String const & );
friend int operator >( String const &, char const * );
friend int operator >( char const *, String const & );
friend int operator >( String const &, char );
friend int operator >( char, String const & );
friend int operator >=( String const &, String const & );
friend int operator >=( String const &, char const * );
friend int operator >=( char const *, String const & );
friend int operator >=( String const &, char );
friend int operator >=( char, String const & );
friend String operator +( String &, String const & );
friend String operator +( String &, char const * );
friend String operator +( char const *, String const & );
friend String operator +( String &, char );
friend String operator +( char, String const & );
friend int valid( String const & );
```

The following I/O Stream inserter and extractor functions are declared:

```
friend istream &operator >>( istream &, String & );
friend ostream &operator <<( ostream &, String const & );
```

## ***String::alloc\_mult\_size()***

---

**Synopsis:**

```
#include <string.hpp>
public:
int String::alloc_mult_size() const;
int String::alloc_mult_size( int mult );
```

**Semantics:** The `alloc_mult_size` public member function is used to query and/or change the allocation multiple size.

The first form of the `alloc_mult_size` public member function queries the current setting.

The second form of the `alloc_mult_size` public member function sets the value to a multiple of 8 based on the *mult* parameter. The value of *mult* is rounded down to a multiple of 8 characters. If *mult* is less than 8, the new multiple size is 1 and allocation sizes are exact.

The scheme used to store a `String` object allocates the memory for the characters in multiples of some size. By default, this size is 8 characters. A `String` object with a length of 10 actually has 16 characters of storage allocated for it. Concatenating more characters on the end of the `String` object only allocates a new storage block if more than 6 (16-10) characters are appended. This scheme tries to find a balance between reallocating frequently (multiples of a small value) and creating a large amount of unused space (multiples of a large value).

**Results:** The `alloc_mult_size` public member function returns the previous allocation multiple size.

**Synopsis:**

```
#include <string.hpp>
public:
char const &String::get_at( size_t pos );
```

**Semantics:** The `get_at` public member function creates a const reference to the character at offset *pos* within the `String` object. This reference may not be used to modify that character. The first character of a `String` object is at position zero.

If *pos* is greater than or equal to the length of the `String` object, and the resulting reference is used, the behavior is undefined.

The reference is associated with the `String` object, and therefore has meaning only as long as the `String` object is not modified (or destroyed). If the `String` object has been modified and an old reference is used, the behavior is undefined.

**Results:** The `get_at` public member function returns a const reference to a character.

**See Also:** `String::put_at`, `operator []`, `operator ()`

## ***String::index()***

---

**Synopsis:**

```
#include <string.hpp>
public:
int String::index( String const &str, size_t pos = 0 ) const;
int String::index( char const *pch, size_t pos = 0 ) const;
```

**Semantics:** The `index` public member function computes the offset at which a sequence of characters in the `String` object is found.

The first form searches the `String` object for the contents of the `str` `String` object.

The second form searches the `String` object for the sequence of characters pointed at by `pch`.

If `pos` is specified, the search begins at that offset from the start of the `String` object. Otherwise, the search begins at offset zero (the first character).

The `index` public member function treats upper and lower case letters as not equal.

**Results:** The `index` public member function returns the offset at which the sequence of characters is found. If the substring is not found, -1 is returned.

**See Also:** `String::lower`, `operator !=`, `operator ==`, `match`, `upper`

**Synopsis:**

```
#include <string.hpp>
public:
size_t String::length() const;
```

**Semantics:** The `length` public member function computes the number of characters contained in the `String` object.

**Results:** The `length` public member function returns the number of characters contained in the `String` object.

## ***String::lower()***

---

**Synopsis:**

```
#include <string.hpp>
public:
String String::lower() const;
```

**Semantics:** The `lower` public member function creates a `String` object whose value is the same as the original object's value, except that all upper-case letters have been converted to lower-case.

**Results:** The `lower` public member function returns a lower-case `String` object.

**See Also:** `String::upper`

**Synopsis:**

```
#include <string.hpp>
public:
int String::match( String const &str ) const;
int String::match( char const *pch ) const;
```

**Semantics:** The `match` public member function compares two character sequences to find the offset where they differ.

The first form compares the `String` object to the *str* `String` object.

The second form compares the `String` object to the *pch* C string.

The first character is at offset zero. The `match` public member function treats upper and lower case letters as not equal.

**Results:** The `match` public member function returns the offset at which the two character sequences differ. If the character sequences are equal, -1 is returned.

**See Also:** `String::index`, `lower`, `operator !=`, `operator ==`, `upper`

## ***String::operator !()***

---

**Synopsis:**

```
#include <string.hpp>
public:
int String::operator !() const;
```

**Semantics:** The operator `!` public member function tests the validity of the `String` object.

**Results:** The operator `!` public member function returns a non-zero value if the `String` object is invalid, otherwise zero is returned.

**See Also:** `String::valid`, `valid`



**Synopsis:**

```
#include <string.hpp>
public:
friend int operator !=( String const &lft,
String const &rht );
friend int operator !=( String const &lft,
char const *rht );
friend int operator !=( char const *lft,
String const &rht );
friend int operator !=( String const &lft,
char rht );
friend int operator !=( char lft,
String const &rht );
```

**Semantics:** The `operator !=` function compares two sequences of characters in terms of an *inequality* relationship.

A `String` object is different from another `String` object if the lengths are different or they contain different sequences of characters. A `String` object and a C string are different if their lengths are different or they contain a different sequence of characters. A C string is terminated by a null character. A `String` object and a character are different if the `String` object does not contain only the character. Upper-case and lower-case characters are considered different.

**Results:** The `operator !=` function returns a non-zero value if the lengths or sequences of characters in the *lft* and *rht* parameter are different, otherwise zero is returned.

**See Also:** `String::operator ==`, `operator <`, `operator <=`, `operator >`, `operator >=`

## ***String::operator ()()***

---

**Synopsis:**

```
#include <string.hpp>
public:
char &String::operator ()( size_t pos );
```

**Semantics:** The `operator ()` public member function creates a reference to the character at offset *pos* within the `String` object. This reference may be used to modify that character. The first character of a `String` object is at position zero.

If *pos* is greater than or equal to the length of the `String` object, and the resulting reference is used, the behavior is undefined.

If the reference is used to modify other characters within the `String` object, the behavior is undefined.

The reference is associated with the `String` object, and therefore has meaning only as long as the `String` object is not modified (or destroyed). If the `String` object has been modified and an old reference is used, the behavior is undefined.

**Results:** The `operator ()` public member function returns a reference to a character.

**See Also:** `String::operator []`, `operator char`, `operator char const *`

**Synopsis:**

```
#include <string.hpp>
public:
String String::operator ()( size_t pos, size_t len ) const;
```

**Semantics:** This form of the `operator ()` public member function extracts a sub-sequence of characters from the `String` object. A new `String` object is created that contains the sub-sequence of characters. The sub-sequence begins at offset *pos* within the `String` object and continues for *len* characters. The first character of a `String` object is at position zero.

If *pos* is greater than or equal to the length of the `String` object, the result is empty.

If *len* is such that *pos + len* exceeds the length of the object, the result is the sub-sequence of characters from the `String` object starting at offset *pos* and running to the end of the `String` object.

**Results:** The `operator ()` public member function returns a `String` object.

**See Also:** `String::operator []`, `operator char`, `operator char const *`

## String operator +()

---

**Synopsis:**

```
#include <string.hpp>
public:
friend String operator +( String &lft,
String const &rht );
friend String operator +( String &lft,
char const *rht );
friend String operator +( char const *lft,
String const &rht );
friend String operator +( String &lft,
char rht );
friend String operator +( char lft,
String const &rht );
```

**Semantics:** The `operator +` function concatenates two sequences of characters into a new `String` object. The new `String` object contains the sequence of characters from the *lft* parameter followed by the sequence of characters from the *rht* parameter.

A `NULL` pointer to a C string is treated as a pointer to an empty C string.

**Results:** The `operator +` function returns a new `String` object that contains the characters from the *lft* parameter followed by the characters from the *rht* parameter.

**See Also:** `String::operator +=`

**Synopsis:**

```
#include <string.hpp>
public:
String &String::operator +=( String const &str );
String &String::operator +=( char const *pch );
```

**Semantics:** The `operator +=` public member function appends the contents of the parameter to the end of the `String` object.

The first form of the `operator +=` public member function appends the contents of the *str* `String` object to the `String` object.

The second form appends the null-terminated sequence of characters stored at *pch* to the `String` object. If the *pch* parameter is `NULL`, nothing is appended.

**Results:** The `operator +=` public member function returns a reference to the `String` object that was the target of the assignment.

**See Also:** `String::operator =`

## String operator <()

---

**Synopsis:**

```
#include <string.hpp>
public:
friend int operator <( String const &lft, String const &rht );
friend int operator <( String const &lft, char const *rht );
friend int operator <( char const *lft, String const &rht );
friend int operator <( String const &lft, char rht );
friend int operator <( char lft, String const &rht );
```

**Semantics:** The `operator <` function compares two sequences of characters in terms of a *less-than* relationship.

*lft* is less-than *rht* if *lft* if the characters of *lft* occur before the characters of *rht* in the collating sequence. Upper-case and lower-case characters are considered different.

**Results:** The `operator <` function returns a non-zero value if the *lft* sequence of characters is less than the *rht* sequence, otherwise zero is returned.

**See Also:** `String::operator !=`, `operator ==`, `operator <=`, `operator >`, `operator >=`

**Synopsis:**

```
#include <string.hpp>
public:
friend ostream &operator <<( ostream &strm,
String const &str );
```

**Semantics:** The operator << function is used to write the sequence of characters in the *str* String object to the *strm* ostream object. Like C strings, the value of the *str* String object is written to *strm* without the addition of any characters. No special processing occurs for any characters in the String object that have special meaning for the *strm* object, such as carriage-returns.

The underlying implementation of the operator << function uses the ostream write method, which writes unformatted characters to the output stream. If formatted output is required, then the programmer should make use of the classes accessor methods, such as `c_str()`, and pass the resulting data item to the stream using the appropriate insert operator.

**Results:** The operator << function returns a reference to the *strm* parameter.

**See Also:** ostream

## String operator <=()

---

**Synopsis:**

```
#include <string.hpp>
public:
friend int operator <=( String const &lft,
String const &rht );
friend int operator <=( String const &lft,
char const *rht );
friend int operator <=( char const *lft,
String const &rht );
friend int operator <=( String const &lft,
char rht );
friend int operator <=( char lft,
String const &rht );
```

**Semantics:** The operator <= function compares two sequences of characters in terms of a *less-than or equal* relationship.

*lft* is less-than or equal to *rht* if the characters of *lft* are equal to or occur before the characters of *rht* in the collating sequence. Upper-case and lower-case characters are considered different.

**Results:** The operator <= function returns a non-zero value if the *lft* sequence of characters is less than or equal to the *rht* sequence, otherwise zero is returned.

**See Also:** `String::operator !=`, `operator ==`, `operator <`, `operator >`, `operator >=`



**Synopsis:**

```
#include <string.hpp>
public:
String &String::operator =( String const &str );
String &String::operator =( char const *pch );
```

**Semantics:** The `operator =` public member function sets the contents of the `String` object to be the same as the parameter.

The first form of the `operator =` public member function sets the value of the `String` object to be the same as the value of the `str` `String` object.

The second form sets the value of the `String` object to the null-terminated sequence of characters stored at `pch`. If the `pch` parameter is `NULL`, the `String` object is empty.

**Results:** The `operator =` public member function returns a reference to the `String` object that was the target of the assignment.

**See Also:** `String::operator +=`, `String`

## String operator ==( )

---

**Synopsis:**

```
#include <string.hpp>
public:
friend int operator ==( String const &lft,
String const &rht );
friend int operator ==( String const &lft,
char const *rht );
friend int operator ==( char const *lft,
String const &rht );
friend int operator ==( String const &lft,
char rht );
friend int operator ==( char lft,
String const &rht );
```

**Semantics:** The operator == function compares two sequences of characters in terms of an *equality* relationship.

A `String` object is equal to another `String` object if they have the same length and they contain the same sequence of characters. A `String` object and a C string are equal if their lengths are the same and they contain the same sequence of characters. The C string is terminated by a null character. A `String` object and a character are equal if the `String` object contains only that character. Upper-case and lower-case characters are considered different.

**Results:** The operator == function returns a non-zero value if the lengths and sequences of characters in the *lft* and *rht* parameter are identical, otherwise zero is returned.

**See Also:** `String::operator !=`, `operator <`, `operator <=`, `operator >`, `operator >=`

**Synopsis:**

```
#include <string.hpp>
public:
friend int operator >( String const &lft, String const &rht );
friend int operator >( String const &lft, char const *rht );
friend int operator >( char const *lft, String const &rht );
friend int operator >( String const &lft, char rht );
friend int operator >( char lft, String const &rht );
```

**Semantics:** The `operator >` function compares two sequences of characters in terms of a *greater-than* relationship.

*lft* is greater-than *rht* if the characters of *lft* occur after the characters of *rht* in the collating sequence. Upper-case and lower-case characters are considered different.

**Results:** The `operator >` function returns a non-zero value if the *lft* sequence of characters is greater than the *rht* sequence, otherwise zero is returned.

**See Also:** `String::operator !=`, `operator ==`, `operator <`, `operator <=`, `operator >=`

## String operator >=()

---

**Synopsis:**

```
#include <string.hpp>
public:
friend int operator >=( String const &lft,
String const &rht );
friend int operator >=( String const &lft,
char const *rht );
friend int operator >=( char const *lft,
String const &rht );
friend int operator >=( String const &lft,
char rht );
friend int operator >=( char lft,
String const &rht );
```

**Semantics:** The operator `>=` function compares two sequences of characters in terms of a *greater-than or equal* relationship.

*lft* is greater-than or equal to *rht* if the characters of *lft* are equal to or occur after the characters of *rht* in the collating sequence. Upper-case and lower-case characters are considered different.

**Results:** The operator `>=` function returns a non-zero value if the *lft* sequence of characters is greater than or equal to the *rht* sequence, otherwise zero is returned.

**See Also:** `String::operator !=`, `operator ==`, `operator <`, `operator <=`, `operator >`

**Synopsis:**

```
#include <string.hpp>
public:
friend istream &operator >>( istream &strm, String &str );
```

**Semantics:** The operator >> function is used to read a sequence of characters from the *strm* istream object into the *str* String object. Like C strings, the gathering of characters for a *str* String object ends at the first whitespace encountered, so that the last character placed in *str* is the character before the whitespace.

**Results:** The operator >> function returns a reference to the *strm* parameter.

**See Also:** istream

## ***String::operator []()***

---

**Synopsis:**

```
#include <string.hpp>
public:
char const &String::operator [] ( size_t pos ) const;
char &String::operator [] ( size_t pos );
```

**Semantics:** The `operator []` public member function creates either a const or a non-const reference to the character at offset *pos* within the `String` object. The non-const reference may be used to modify that character. The first character of a `String` object is at position zero.

If *pos* is greater than or equal to the length of the `String` object, and the resulting reference is used, the behavior is undefined.

If the non-const reference is used to modify other characters within the `String` object, the behavior is undefined.

The reference is associated with the `String` object, and therefore has meaning only as long as the `String` object is not modified (or destroyed). If the `String` object has been modified and an old reference is used, the behavior is undefined.

**Results:** The `operator []` public member function returns either a const or a non-const reference to a character.

**See Also:** `String::operator ()`, `operator char`, `operator char const *`

**Synopsis:**

```
#include <string.hpp>
public:
String::operator char();
```

**Semantics:** The `operator char` public member function converts a `String` object into the first character it contains. If the `String` object is empty, the result is the null character.

**Results:** The `operator char` public member function returns the first character contained in the `String` object. If the `String` object is empty, the null character is returned.

**See Also:** `String::operator ()`, `operator []`, `operator char const *`

## ***String::operator char const \*()***

---

**Synopsis:**

```
#include <string.hpp>
public:
String::operator char const *();
```

**Semantics:** The `operator char const *` public member function converts a `String` object into a C string containing the same length and sequence of characters, terminated by a null character. If the `String` object contains a null character the resulting C string is terminated by that null character.

The returned pointer is associated with the `String` object, and therefore has meaning only as long as the `String` object is not modified. If the intention is to be able to refer to the C string after the `String` object has been modified, a copy of the string should be made, perhaps by using the C library `strdup` function.

The returned pointer is a pointer to a constant C string. If the pointer is used in some way to modify the C string, the behavior is undefined.

**Results:** The `operator char const *` public member function returns a pointer to a null-terminated constant C string that contains the same characters as the `String` object.

**See Also:** `String::operator ()`, `operator []`, `operator char`



**Synopsis:**

```
#include <string.hpp>
public:
void String::put_at( size_t pos, char chr );
```

**Semantics:** The `put_at` public member function modifies the character at offset *pos* within the `String` object. The character at the specified offset is set to the value of *chr*. If *pos* is greater than the number of characters within the `String` object, *chr* is appended to the `String` object.

**Results:** The `put_at` public member function has no return value.

**See Also:** `String::get_at`, `operator []`, `operator ()`, `operator +=`, `operator +`

## ***String::String()***

---

**Synopsis:**

```
#include <string.hpp>
public:
String::String();
```

**Semantics:** This form of the public `String` constructor creates a default `String` object containing no characters. The created `String` object has length zero.

**Results:** This form of the public `String` constructor produces a `String` object.

**See Also:** `String::operator =`, `operator +=`, `~String`

**Synopsis:**

```
#include <string.hpp>
public:
String::String( size_t size, String::capacity cap );
```

**Semantics:** This form of the public `String` constructor creates a `String` object. The function constructs a `String` object of length *size* if *cap* is equal to the enumerated *default\_size*. The function reserves *size* bytes of memory and sets the length of the `String` object to be zero if *cap* is equal to the enumerated *reserve*.

**Results:** This form of the public `String` constructor produces a `String` object of size *size*.

**See Also:** `String::operator =`, `~String`

## ***String::String()***

---

**Synopsis:**

```
#include <string.hpp>
public:
String::String( String const &str,
size_t pos = 0,
size_t num = NPOS );
```

**Semantics:** This form of the public `String` constructor creates a `String` object which contains a sub-string of the `str` parameter. The sub-string starts at position `pos` within `str` and continues for `num` characters or until the end of the `str` parameter, whichever comes first.

**Results:** This form of the public `String` constructor produces a sub-string or duplicate of the `str` parameter.

**See Also:** `String::operator =`, `operator ()`, `operator []`, `~String`

**Synopsis:**

```
#include <string.hpp>
public:
String::String( char const *pch, size_t num = NPOS );
```

**Semantics:** This form of the public `String` constructor creates a `String` object from a C string. The `String` object contains the sequence of characters located at the `pch` parameter. Characters are included up to `num` or the end of the C string pointed at by `pch`. Note that C strings are terminated by a null character and that the value of the created `String` object does not contain that character, nor any following it.

**Results:** This form of the public `String` constructor produces a `String` object of at most length `n` containing the characters in the C string starting at the `pch` parameter.

**See Also:** `String::operator =`, `operator char const *`, `operator ()`, `operator []`, `~String`

## ***String::String()***

---

**Synopsis:**

```
#include <string.hpp>
public:
String::String( char ch, size_t rep = 1 );
```

**Semantics:** This form of the public `String` constructor creates a `String` object containing *rep* copies of the *ch* parameter.

**Results:** This form of the public `String` constructor produces a `String` object of length *rep* containing only the character specified by the *ch* parameter.

**See Also:** `String::operator =`, `operator char`, `~String`

**Synopsis:**

```
#include <string.hpp>
public:
String::~~String();
```

**Semantics:** The public `~String` destructor destroys the `String` object. The call to the public `~String` destructor is inserted implicitly by the compiler at the point where the `String` object goes out of scope.

**Results:** The `String` object is destroyed.

**See Also:** `String`

## ***String::upper()***

---

**Synopsis:**

```
#include <string.hpp>
public:
String String::upper() const;
```

**Semantics:** The `upper` public member function creates a new `String` object whose value is the same as the original `String` object, except that all lower-case letters have been converted to upper-case.

**Results:** The `upper` public member function returns a new upper-case `String` object.

**See Also:** `String::lower`



**Synopsis:**

```
#include <string.hpp>
public:
friend int valid( String const &str );
```

**Semantics:** The `valid` function tests the validity of the *str* `String` object.

**Results:** The `valid` function returns a non-zero value if the *str* `String` object is valid, otherwise zero is returned.

**See Also:** `String::operator !`, `valid`

## ***String::valid()***

---

**Synopsis:**

```
#include <string.hpp>
public:
int String::valid() const;
```

**Semantics:** The `valid` public member function tests the validity of the `String` object.

**Results:** The `valid` public member function returns a non-zero value if the `String` object is valid, otherwise zero is returned.

**See Also:** `String::operator !`, `valid`



\_\_NOT\_EOF 9



abs, related function  
 Complex 21, 23  
 acos, related function  
 Complex 21, 24  
 acosh, related function  
 Complex 21, 25  
 adjustfield, member enumeration  
 ios 683  
 all\_fine, member enumeration  
 WCEexcept 74  
 WCIterExcept 80  
 alloc\_mult\_size, member function  
 String 886, 888  
 alloc\_size\_increment, member function  
 strstreambuf 867, 869  
 allocate, member function  
 streambuf 813, 815  
 allocator  
 function 267, 271, 295, 299, 431, 531  
 app, member enumeration  
 ios 694  
 append, member enumeration  
 ios 694  
 append, member function  
 WCIsvConstDListIter<Type> 318  
 WCIsvConstSListIter<Type> 318  
 WCIsvDList<Type> 241, 249  
 WCIsvDListIter<Type> 335, 343  
 WCIsvSList<Type> 241, 249  
 WCIsvSListIter<Type> 335, 343  
 WCPtrConstDListIter<Type> 354

WCPtrConstSListIter<Type> 354  
 WCPtrDList<Type> 264, 274  
 WCPtrDListIter<Type> 371, 379  
 WCPtrOrderedVector<Type> 542, 549  
 WCPtrSList<Type> 264, 274  
 WCPtrSListIter<Type> 371, 379  
 WCPtrSortedVector<Type> 542, 549  
 WCValConstDListIter<Type> 390  
 WCValConstSListIter<Type> 390  
 WCValDList<Type> 291, 302  
 WCValDListIter<Type> 407, 416  
 WCValOrderedVector<Type> 586, 593  
 WCValSList<Type> 291, 302  
 WCValSListIter<Type> 407, 416  
 WCValSortedVector<Type> 586, 593  
 arg, related function  
 Complex 21, 26  
 asin, related function  
 Complex 21, 27  
 asinh, related function  
 Complex 21, 28  
 atan, related function  
 Complex 21, 29  
 atanh, related function  
 Complex 21, 30  
 ate, member enumeration  
 ios 694  
 atend, member enumeration  
 ios 694  
 attach, member function  
 filebuf 629-630  
 fstreambase 653-654



bad, member function  
 ios 673, 675  
 badbit, member enumeration  
 ios 692  
 base, member function

- streambuf 813, 816
- basefield, member enumeration
  - ios 683
- beg, member enumeration
  - ios 702
- binary, member enumeration
  - ios 694
- bitalloc, member function
  - ios 674, 676
- bitHash, member function
  - WCPtrHashDict<Key, Value> 89, 94
  - WCPtrHashSet<Type> 112, 121
  - WCPtrHashTable<Type> 112, 121
  - WCValHashDict<Key, Value> 138, 143
  - WCValHashSet<Type> 160, 169
  - WCValHashTable<Type> 160, 169
- blen, member function
  - streambuf 813, 817
- buckets, member function
  - WCPtrHashDict<Key, Value> 89, 95
  - WCPtrHashSet<Type> 112, 122
  - WCPtrHashTable<Type> 112, 122
  - WCValHashDict<Key, Value> 138, 144
  - WCValHashSet<Type> 160, 170
  - WCValHashTable<Type> 160, 170

## C

- cerr 11
- check\_all, member enumeration
  - WCExcept 74
  - WCIterExcept 80
- check\_none, member enumeration
  - WCExcept 74
  - WCIterExcept 80
- cin 11
- clear, member function
  - ios 673, 677
  - WCIsvDList<Type> 241, 250
  - WCIsvSList<Type> 241, 250

- WCPtrDList<Type> 264, 275
- WCPtrHashDict<Key, Value> 89, 96
- WCPtrHashSet<Type> 112, 123
- WCPtrHashTable<Type> 112, 123
- WCPtrOrderedVector<Type> 541, 550
- WCPtrSkipList<Type> 463, 472
- WCPtrSkipListDict<Key, Value> 443, 448
- WCPtrSkipListSet<Type> 463, 472
- WCPtrSList<Type> 264, 275
- WCPtrSortedVector<Type> 541, 550
- WCPtrVector<Type> 572, 577
- WCQueue<Type, FType> 428, 433
- WCStack<Type, FType> 528, 533
- WCValDList<Type> 291, 303
- WCValHashDict<Key, Value> 138, 145
- WCValHashSet<Type> 160, 171
- WCValHashTable<Type> 160, 171
- WCValOrderedVector<Type> 585, 594
- WCValSkipList<Type> 505, 514
- WCValSkipListDict<Key, Value> 486, 491
- WCValSkipListSet<Type> 505, 514
- WCValSList<Type> 291, 303
- WCValSortedVector<Type> 585, 594
- WCValVector<Type> 616, 621
- clearAndDestroy, member function
  - WCIsvDList<Type> 241, 251
  - WCIsvSList<Type> 241, 251
  - WCPtrDList<Type> 264, 276
  - WCPtrHashDict<Key, Value> 89, 97
  - WCPtrHashSet<Type> 112, 124
  - WCPtrHashTable<Type> 112, 124
  - WCPtrOrderedVector<Type> 541, 551
  - WCPtrSkipList<Type> 463, 473
  - WCPtrSkipListDict<Key, Value> 443, 449
  - WCPtrSkipListSet<Type> 463, 473
  - WCPtrSList<Type> 264, 276
  - WCPtrSortedVector<Type> 541, 551
  - WCPtrVector<Type> 572, 578
  - WCValDList<Type> 291, 304
  - WCValSList<Type> 291, 304
- clog 11
- close, member function
  - filebuf 629, 631
  - fstreambase 653, 655

- common types 9
- Complex class 19
- Complex related functions
  - abs 21, 23
  - acos 21, 24
  - acosh 21, 25
  - arg 21, 26
  - asin 21, 27
  - asinh 21, 28
  - atan 21, 29
  - atanh 21, 30
  - conj 21, 35
  - cos 21, 36
  - cosh 21, 37
  - exp 21, 38
  - imag 21, 40
  - log 21
  - log10 21, 42
  - norm 21, 43
  - num 41
  - operator != 21, 44
  - operator \* 21, 45
  - operator + 20-21, 48
  - operator - 21, 51
  - operator / 21, 53
  - operator << 20, 55
  - operator == 21, 57
  - operator >> 20, 58
  - polar 21, 59
  - pow 21, 60
  - real 21, 62
  - sin 21, 63
  - sinh 21, 64
  - sqrt 21, 65
  - tan 22, 66
  - tanh 22, 67
- Complex::Complex 20, 31-33
- Complex::imag 20, 39
- Complex::operator \*= 20, 46
- Complex::operator + 20, 47
- Complex::operator += 20, 49
- Complex::operator - 20, 50
- Complex::operator -= 20, 52
- Complex::operator /= 20, 54
- Complex::operator = 20, 56
- Complex::real 20, 61
- Complex::~Complex 20, 34
- conj, related function
  - Complex 21, 35
- constructor
  - Complex 20, 31-33
  - filebuf 629, 633-635
  - fstream 646-650
  - fstreambase 653, 656-659
  - ifstream 666-670
  - ios 673, 689-690
  - iostream 710-713
  - istream 717, 729-731
  - istrstream 748-750
  - ofstream 767-771
  - ostream 774, 789-791
  - ostrstream 798-800
  - stdiobuf 804, 806-807
  - streambuf 813, 851-852
  - String 886, 912-916
  - strstream 857, 859-860
  - strstreambase 862, 864-865
  - strstreambuf 867, 876-879
  - WCDLink 237-238
  - WCEXcept 70-71
  - WCISvConstSListIter<Type> 318
  - WCISvSList<Type> 240-241
  - WCISvSListIter<Type> 335
  - WCIterExcept 76-77
  - WCPtrConstSListIter<Type> 354
  - WCPtrHashDict<Key, Value> 90-92
  - WCPtrHashDictIter<Key, Value> 186
  - WCPtrHashSetIter<Type> 208
  - WCPtrHashTable<Type> 113-115, 117-119
  - WCPtrSkipList<Type> 464-466, 468-470
  - WCPtrSkipListDict<Key, Value> 444-446
  - WCPtrSList<Type> 264
  - WCPtrSListIter<Type> 371
  - WCPtrSortedVector<Type> 543-544, 546-547
  - WCPtrVector<Type> 572-575
  - WCQueue<Type, FType> 428, 430-431
  - WCSTLink 288-289
  - WCStack<Type, FType> 528, 530-531

- WCValConstSListIter<Type> 390
- WCValHashDict<Key,Value> 139-141
- WCValHashDictIter<Key,Value> 197
- WCValHashSetIter<Type> 221
- WCValHashTable<Type> 161-163, 165-167
- WCValSkipList<Type> 506-508, 510-512
- WCValSkipListDict<Key,Value> 487-489
- WCValSList<Type> 291
- WCValSListIter<Type> 407
- WCValSortedVector<Type> 587-588, 590-591
- WCValVector<Type> 615, 617-619
- container, member function
  - WCIsvConstDListIter<Type> 318, 326
  - WCIsvConstSListIter<Type> 318, 326
  - WCIsvDListIter<Type> 335, 344
  - WCIsvSListIter<Type> 335, 344
  - WCPtrConstDListIter<Type> 354, 362
  - WCPtrConstSListIter<Type> 354, 362
  - WCPtrDListIter<Type> 371, 380
  - WCPtrHashDictIter<Key,Value> 186, 190
  - WCPtrHashSetIter<Type> 208, 215
  - WCPtrHashTableIter<Type> 208, 215
  - WCPtrSListIter<Type> 371, 380
  - WCValConstDListIter<Type> 390, 398
  - WCValConstSListIter<Type> 390, 398
  - WCValDListIter<Type> 407, 417
  - WCValHashDictIter<Key,Value> 197, 201
  - WCValHashSetIter<Type> 221, 228
  - WCValHashTableIter<Type> 221, 228
  - WCValSListIter<Type> 407, 417
- contains, member function
  - WCIsvDList<Type> 241, 252
  - WCIsvSList<Type> 241, 252
  - WCPtrDList<Type> 264, 277
  - WCPtrHashDict<Key,Value> 89, 98
  - WCPtrHashSet<Type> 112, 125
  - WCPtrHashTable<Type> 112, 125
  - WCPtrOrderedVector<Type> 541, 552
  - WCPtrSkipList<Type> 463, 474
  - WCPtrSkipListDict<Key,Value> 443, 450
  - WCPtrSkipListSet<Type> 463, 474
  - WCPtrSList<Type> 264, 277
  - WCPtrSortedVector<Type> 541, 552
- WCValDList<Type> 291, 305
- WCValHashDict<Key,Value> 138, 146
- WCValHashSet<Type> 160, 172
- WCValHashTable<Type> 160, 172
- WCValOrderedVector<Type> 585, 595
- WCValSkipList<Type> 505, 515
- WCValSkipListDict<Key,Value> 486, 492
- WCValSkipListSet<Type> 505, 515
- WCValSList<Type> 291, 305
- WCValSortedVector<Type> 585, 595
- cos, related function
  - Complex 21, 36
- cosh, related function
  - Complex 21, 37
- cout 11
- cur, member enumeration
  - ios 702
- current, member function
  - WCIsvConstDListIter<Type> 318, 327
  - WCIsvConstSListIter<Type> 318, 327
  - WCIsvDListIter<Type> 335, 345
  - WCIsvSListIter<Type> 335, 345
  - WCPtrConstDListIter<Type> 354, 363
  - WCPtrConstSListIter<Type> 354, 363
  - WCPtrDListIter<Type> 371, 381
  - WCPtrHashSetIter<Type> 208, 216
  - WCPtrHashTableIter<Type> 208, 216
  - WCPtrSListIter<Type> 371, 381
  - WCValConstDListIter<Type> 390, 399
  - WCValConstSListIter<Type> 390, 399
  - WCValDListIter<Type> 407, 418
  - WCValHashSetIter<Type> 221, 229
  - WCValHashTableIter<Type> 221, 229
  - WCValSListIter<Type> 407, 418

D

- dbp, member function
  - streambuf 813, 818
- dealloc

function 267, 271, 295, 299, 431, 531  
 dec, manipulator 752-753  
 dec, member enumeration  
   ios 683  
 destructor  
   Complex 20, 34  
   filebuf 629, 636  
   fstream 646, 651  
   fstreambase 653, 660  
   ifstream 666, 671  
   ios 673, 691  
   iostream 710, 714  
   istream 717, 732  
   istrstream 748, 751  
   ofstream 767, 772  
   ostream 774, 792  
   ostrstream 798, 801  
   stdiobuf 804, 808  
   streambuf 813, 853  
   String 886, 917  
   strstream 857, 861  
   strstreambase 862, 866  
   strstreambuf 867, 881  
   WCDLink 237, 239  
   WCEexcept 70, 72  
   WCIsvConstSListIter<Type> 318  
   WCIsvSList<Type> 240-241  
   WCIsvSListIter<Type> 335  
   WCIterExcept 76, 78  
   WCPtrConstSListIter<Type> 354  
   WCPtrHashDict<Key, Value> 93  
   WCPtrHashDictIter<Key, Value> 186  
   WCPtrHashSetIter<Type> 208  
   WCPtrHashTable<Type> 116, 120  
   WCPtrSkipList<Type> 467, 471  
   WCPtrSkipListDict<Key, Value> 447  
   WCPtrSList<Type> 264  
   WCPtrSListIter<Type> 371  
   WCPtrSortedVector<Type> 545, 548  
   WCPtrVector<Type> 572, 576  
   WCQueue<Type, FType> 428, 432  
   WCSLink 288, 290  
   WCStack<Type, FType> 528, 532  
   WCValConstSListIter<Type> 390

WCValHashDict<Key, Value> 142  
 WCValHashDictIter<Key, Value> 197  
 WCValHashSetIter<Type> 221  
 WCValHashTable<Type> 164, 168  
 WCValSkipList<Type> 509, 513  
 WCValSkipListDict<Key, Value> 490  
 WCValSList<Type> 291  
 WCValSListIter<Type> 407  
 WCValSortedVector<Type> 589, 592  
 WCValVector<Type> 615, 620  
 do\_sgetn, member function  
   streambuf 813, 819  
 do\_sputn, member function  
   streambuf 813, 820  
 doallocate, member function  
   streambuf 813, 821  
   strstreambuf 867, 870

E

eatwhite, member function  
   istream 717, 719  
 eback, member function  
   streambuf 813, 822  
 ebuf, member function  
   streambuf 813, 823  
 egptr, member function  
   streambuf 813, 824  
 empty\_container  
   exception 74, 254-255, 257, 279-280, 282,  
     307-308, 310, 435-436, 439, 536, 538,  
     555, 560, 562, 568-570, 580, 598, 603,  
     605, 611-613, 623  
 empty\_container, member enumeration  
   WCEexcept 74  
 end, member enumeration  
   ios 702  
 endl, manipulator 752, 754  
 ends, manipulator 752, 755  
 entries, member function

WCIsvDList<Type> 241, 253  
 WCIsvSList<Type> 241, 253  
 WCPtrDList<Type> 264, 278  
 WCPtrHashDict<Key,Value> 89, 99  
 WCPtrHashSet<Type> 112, 126  
 WCPtrHashTable<Type> 112, 126  
 WCPtrOrderedVector<Type> 541, 553  
 WCPtrSkipList<Type> 463, 475  
 WCPtrSkipListDict<Key,Value> 443, 451  
 WCPtrSkipListSet<Type> 463, 475  
 WCPtrSList<Type> 264, 278  
 WCPtrSortedVector<Type> 541, 553  
 WCQueue<Type,FType> 428, 434  
 WCStack<Type,FType> 528, 534  
 WCValDList<Type> 292, 306  
 WCValHashDict<Key,Value> 138, 147  
 WCValHashSet<Type> 160, 173  
 WCValHashTable<Type> 160, 173  
 WCValOrderedVector<Type> 585, 596  
 WCValSkipList<Type> 505, 516  
 WCValSkipListDict<Key,Value> 486, 493  
 WCValSkipListSet<Type> 505, 516  
 WCValSList<Type> 292, 306  
 WCValSortedVector<Type> 585, 596  
 EOF 9  
 eof, member function  
     ios 674, 678  
 eofbit, member enumeration  
     ios 692  
 epptr, member function  
     streambuf 813, 825  
 exception handling 4  
 exceptions 74  
     function 80  
 exceptions, member function  
     ios 674, 679  
     WCExcept 70, 73  
     WCIterExcept 76, 79  
 exp, related function  
     Complex 21, 38  
 extractor 13, 717

**F**

fail, member function  
     ios 673, 680  
 failbit, member enumeration  
     ios 692  
 fd, member function  
     filebuf 629, 632  
     fstreambase 653, 662  
 filebuf 811  
 filebuf::attach 629-630  
 filebuf::close 629, 631  
 filebuf::fd 629, 632  
 filebuf::filebuf 629, 633-635  
 filebuf::is\_open 629, 637  
 filebuf::open 629, 638  
 filebuf::openprot 628, 639  
 filebuf::overflow 629, 640  
 filebuf::pbackfail 629, 641  
 filebuf::seekoff 629, 642  
 filebuf::setbuf 629, 643  
 filebuf::sync 629, 644  
 filebuf::underflow 629, 645  
 filebuf::~filebuf 629, 636  
 filedesc 9  
 fill character 681  
 fill, member function  
     ios 674, 681  
 find, member function  
     WCIsvDList<Type> 241, 254  
     WCIsvSList<Type> 241, 254  
     WCPtrDList<Type> 264, 279  
     WCPtrHashDict<Key,Value> 89, 100  
     WCPtrHashSet<Type> 112, 127  
     WCPtrHashTable<Type> 112, 127  
     WCPtrOrderedVector<Type> 541, 554  
     WCPtrSkipList<Type> 463, 476  
     WCPtrSkipListDict<Key,Value> 443, 452  
     WCPtrSkipListSet<Type> 463, 476  
     WCPtrSList<Type> 264, 279



- WCPtrSortedVector<Type> 541, 554
- WCValDList<Type> 292, 307
- WCValHashDict<Key, Value> 138, 148
- WCValHashSet<Type> 160, 174
- WCValHashTable<Type> 160, 174
- WCValOrderedVector<Type> 585, 597
- WCValSkipList<Type> 505, 517
- WCValSkipListDict<Key, Value> 486, 494
- WCValSkipListSet<Type> 505, 517
- WCValSList<Type> 292, 307
- WCValSortedVector<Type> 585, 597
- findKeyAndValue, member function
  - WCPtrHashDict<Key, Value> 89, 101
  - WCPtrSkipListDict<Key, Value> 443, 453
  - WCValHashDict<Key, Value> 138, 149
  - WCValSkipListDict<Key, Value> 486, 495
- findLast, member function
  - WCIsvDList<Type> 241, 255
  - WCIsvSList<Type> 241, 255
  - WCPtrDList<Type> 264, 280
  - WCPtrSList<Type> 264, 280
  - WCValDList<Type> 292, 308
  - WCValSList<Type> 292, 308
- first, member function
  - WCPtrOrderedVector<Type> 541, 555
  - WCPtrSortedVector<Type> 541, 555
  - WCQueue<Type, FType> 428, 435
  - WCValOrderedVector<Type> 586, 598
  - WCValSortedVector<Type> 586, 598
- fixed, member enumeration
  - ios 683
- flags, member function
  - ios 674, 682
- floatfield, member enumeration
  - ios 683
- flush, manipulator 752, 756
- flush, member function
  - ostream 774, 776
- fmtflags, member enumeration
  - ios 673, 683
- forall, member function
  - WCIsvDList<Type> 241, 256
  - WCIsvSList<Type> 241, 256
  - WCPtrDList<Type> 264, 281
  - WCPtrHashDict<Key, Value> 89, 102
  - WCPtrHashSet<Type> 112, 128
  - WCPtrHashTable<Type> 112, 128
  - WCPtrSkipList<Type> 463, 477
  - WCPtrSkipListDict<Key, Value> 443, 454
  - WCPtrSkipListSet<Type> 463, 477
  - WCPtrSList<Type> 264, 281
  - WCValDList<Type> 292, 309
  - WCValHashDict<Key, Value> 138, 150
  - WCValHashSet<Type> 160, 175
  - WCValHashTable<Type> 160, 175
  - WCValSkipList<Type> 505, 518
  - WCValSkipListDict<Key, Value> 486, 496
  - WCValSkipListSet<Type> 505, 518
  - WCValSList<Type> 292, 309
- format precision 698
- format width 708
- formatted input 13
- formatted output 15
- freeze, member function
  - strstreambuf 867, 871
- fstream 653, 710
- fstream::fstream 646-650
- fstream::open 646, 652
- fstream::~fstream 646, 651
- fstreambase 646, 666, 767
- fstreambase::attach 653-654
- fstreambase::close 653, 655
- fstreambase::fd 653, 662
- fstreambase::fstreambase 653, 656-659
- fstreambase::is\_open 653, 661
- fstreambase::open 653, 663
- fstreambase::rdbuf 653, 664
- fstreambase::setbuf 653, 665
- fstreambase::~fstreambase 653, 660
- functions and types 17

G

gbump, member function

- streambuf 813, 826
- gcount, member function
  - istream 718, 720
- get area 811
- get pointer 827
- get, member function
  - istream 717-718, 721-724
    - WCIsvDList<Type> 241, 257
    - WCIsvSList<Type> 241, 257
    - WCPtrDList<Type> 264, 282
    - WCPtrSList<Type> 264, 282
    - WCQueue<Type,FType> 428, 436
    - WCValDList<Type> 292, 310
    - WCValSList<Type> 292, 310
- get\_at, member function
  - String 886, 889
- getline, member function
  - istream 718, 725
- good, member function
  - ios 673, 687
- goodbit, member enumeration
  - ios 692
- gptr, member function
  - streambuf 813, 827

- limits 5
- list 5
- map 5
- memory 5
- new 5
- numeric 5
- ostream 5
- set 5
- stdiobuf 6
- streambuf 6
- string 6
- strstream 6
- vector 6
- wcdefs 6
- wclbase 6
- wclcom 6
- wclibase 6
- wclist 6
- wclistit 6
- wcqueue 7
- wcstack 7
- hex, manipulator 752, 757
- hex, member enumeration
  - ios 683

## H

- header files
  - algorithm 3
  - complex 4
  - exception 4
  - fstream 4
  - functional 4
  - generic 4
  - iomanip 4
  - ios 4
  - iosfwd 4
  - iostream 4
  - istream 5
  - iterator 5

## I

- ifstream 653, 717
- ifstream::ifstream 666-670
- ifstream::open 666, 672
- ifstream::~ifstream 666, 671
- ignore, member function
  - istream 718, 726
- imag, member function
  - Complex 20, 39
- imag, related function
  - Complex 21, 40
- in, member enumeration
  - ios 694
- in\_avail, member function

- streambuf 813, 828
- index, member function
  - String 886, 890
  - WCIsvDList<Type> 241, 258-259
  - WCIsvSList<Type> 241, 258-259
  - WCPtrDList<Type> 264, 283
  - WCPtrOrderedVector<Type> 541, 556
  - WCPtrSList<Type> 264, 283
  - WCPtrSortedVector<Type> 541, 556
  - WCValDList<Type> 292, 311
  - WCValOrderedVector<Type> 586, 599
  - WCValSList<Type> 292, 311
  - WCValSortedVector<Type> 586, 599
- index\_range
  - exception 74, 106, 154, 255, 280, 308, 435-436, 439, 458, 500, 536, 538, 555, 558, 560, 562, 580, 598, 601, 603, 605, 623
- index\_range, member enumeration
  - WCExcept 74
- init, member function
  - ios 673, 688
- insert, member function
  - WCIsvConstDListIter<Type> 318
  - WCIsvConstSListIter<Type> 318
  - WCIsvDList<Type> 241, 260
  - WCIsvDListIter<Type> 335-336, 346
  - WCIsvSList<Type> 241, 260
  - WCIsvSListIter<Type> 335-336, 346
  - WCPtrConstDListIter<Type> 354
  - WCPtrConstSListIter<Type> 354
  - WCPtrDList<Type> 264, 284
  - WCPtrDListIter<Type> 371-372, 382
  - WCPtrHashDict<Key, Value> 89, 103
  - WCPtrHashSet<Type> 112, 129
  - WCPtrHashTable<Type> 112, 129
  - WCPtrOrderedVector<Type> 541, 557
  - WCPtrSkipList<Type> 463, 478
  - WCPtrSkipListDict<Key, Value> 443, 455
  - WCPtrSkipListSet<Type> 463, 478
  - WCPtrSList<Type> 264, 284
  - WCPtrSListIter<Type> 371-372, 382
  - WCPtrSortedVector<Type> 541, 557
  - WCQueue<Type, FType> 428, 437
  - WCValConstDListIter<Type> 390
  - WCValConstSListIter<Type> 390
  - WCValDList<Type> 292, 312
  - WCValDListIter<Type> 407-408, 419
  - WCValHashDict<Key, Value> 138, 151
  - WCValHashSet<Type> 160, 176
  - WCValHashTable<Type> 160, 176
  - WCValOrderedVector<Type> 586, 600
  - WCValSkipList<Type> 505, 519
  - WCValSkipListDict<Key, Value> 486, 497
  - WCValSkipListSet<Type> 505, 519
  - WCValSList<Type> 292, 312
  - WCValSListIter<Type> 407-408, 419
  - WCValSortedVector<Type> 586, 600
- insertAt, member function
  - WCPtrOrderedVector<Type> 542, 558
  - WCPtrSortedVector<Type> 542, 558
  - WCValOrderedVector<Type> 586, 601
  - WCValSortedVector<Type> 586, 601
- inserter 15, 774
- internal, member enumeration
  - ios 683
- intrusive
  - classes 240
- ios 653, 717, 774, 862
  - ios::adjustfield 683
  - ios::app 694
  - ios::append 694
  - ios::ate 694
  - ios::atend 694
  - ios::bad 673, 675
  - ios::badbit 692
  - ios::basefield 683
  - ios::beg 702
  - ios::binary 694
  - ios::bitalloc 674, 676
  - ios::clear 673, 677
  - ios::cur 702
  - ios::dec 683
  - ios::end 702
  - ios::eof 674, 678
  - ios::eofbit 692
  - ios::exceptions 674, 679
  - ios::fail 673, 680

ios::failbit 692  
ios::fill 674, 681  
ios::fixed 683  
ios::flags 674, 682  
ios::floatfield 683  
ios::fmtflags 673, 683  
ios::good 673, 687  
ios::goodbit 692  
ios::hex 683  
ios::in 694  
ios::init 673, 688  
ios::internal 683  
ios::ios 673, 689-690  
ios::iostate 673, 692  
ios::iword 674, 693  
ios::left 683  
ios::nocreate 694  
ios::noreplace 694  
ios::oct 683  
ios::openmode 673, 694  
ios::operator ! 674, 696  
ios::operator void \* 674, 697  
ios::out 694  
ios::precision 674, 698  
ios::pword 674, 699  
ios::rdbuf 673, 700  
ios::rdstate 673, 701  
ios::right 683  
ios::scientific 683  
ios::seekdir 673, 702  
ios::setf 674, 703  
ios::setstate 673, 704  
ios::showbase 683  
ios::showpoint 683  
ios::showpos 683  
ios::skipws 683  
ios::stdio 683  
ios::sync\_with\_stdio 674, 705  
ios::text 694  
ios::tie 673, 706  
ios::trunc 694  
ios::truncate 694  
ios::unitbuf 683  
ios::unsetf 674, 707  
ios::uppercase 683  
ios::width 674, 708  
ios::xalloc 674, 709  
ios::~ios 673, 691  
iostate, member enumeration  
    ios 673, 692  
iostream 646, 717, 774, 857  
iostream::iostream 710-713  
iostream::operator = 710, 715-716  
iostream::~iostream 710, 714  
ipfx, member function  
    istream 717, 727  
is\_open, member function  
    filebuf 629, 637  
    fstreambase 653, 661  
isEmpty, member function  
    WCIsvDList<Type> 241, 261  
    WCIsvSList<Type> 241, 261  
    WCPtrDList<Type> 264, 285  
    WCPtrHashDict<Key, Value> 89, 104  
    WCPtrHashSet<Type> 112, 130  
    WCPtrHashTable<Type> 112, 130  
    WCPtrOrderedVector<Type> 541, 559  
    WCPtrSkipList<Type> 463, 479  
    WCPtrSkipListDict<Key, Value> 443, 456  
    WCPtrSkipListSet<Type> 463, 479  
    WCPtrSList<Type> 264, 285  
    WCPtrSortedVector<Type> 541, 559  
    WCQueue<Type, FType> 428, 438  
    WCStack<Type, FType> 528, 535  
    WCValDList<Type> 292, 313  
    WCValHashDict<Key, Value> 138, 152  
    WCValHashSet<Type> 160, 177  
    WCValHashTable<Type> 160, 177  
    WCValOrderedVector<Type> 586, 602  
    WCValSkipList<Type> 505, 520  
    WCValSkipListDict<Key, Value> 486, 498  
    WCValSkipListSet<Type> 505, 520  
    WCValSList<Type> 292, 313  
    WCValSortedVector<Type> 586, 602  
isfx, member function  
    istream 717, 728  
istream 666, 673, 710, 748  
istream input 13

istream::eatwhite 717, 719  
 istream::gcount 718, 720  
 istream::get 717-718, 721-724  
 istream::getline 718, 725  
 istream::ignore 718, 726  
 istream::ipfx 717, 727  
 istream::isfx 717, 728  
 istream::istream 717, 729-731  
 istream::operator = 718, 733-734  
 istream::operator >> 718, 735-740  
 istream::peek 718, 741  
 istream::putback 718, 742  
 istream::read 718, 743  
 istream::seekg 718, 744-745  
 istream::sync 718, 746  
 istream::tellg 718, 747  
 istream::~istream 717, 732  
 istrstream 717, 862  
 istrstream::istrstream 748-750  
 istrstream::~istrstream 748, 751  
 iter\_range  
     exception 80, 330, 332, 349, 351, 366, 368,  
         385, 387, 402, 404, 422, 424  
 iter\_range, member enumeration  
     WCIterExcept 80  
 iterator classes 6  
 iword, member function  
     ios 674, 693

## K

key, member function  
     WCPtrHashDictIter<Key,Value> 186, 191  
     WCValHashDictIter<Key,Value> 197, 202

## L

last, member function  
     WCPtrOrderedVector<Type> 541, 560  
     WCPtrSortedVector<Type> 541, 560  
     WCQueue<Type,FType> 428, 439  
     WCValOrderedVector<Type> 586, 603  
     WCValSortedVector<Type> 586, 603  
 left, member enumeration  
     ios 683  
 length, member function  
     String 886, 891  
     WCPtrVector<Type> 572, 579  
     WCValVector<Type> 616, 622  
 list containers 6  
 log, related function  
     Complex 21  
 log10, related function  
     Complex 21, 42  
 lower, member function  
     String 886, 892

## M

manipulator manipulators  
     dec 752-753  
     endl 752, 754  
     ends 752, 755  
     flush 752, 756  
     hex 752, 757  
     oct 752, 758  
     resetiosflags 752, 759  
     setbase 752, 760  
     setfill 752, 761  
     setiosflags 752, 762  
     setprecision 752, 763  
     setw 752, 764

- setwidth 752, 765
- ws 752, 766
- manipulators
  - dec 752-753
  - endl 752, 754
  - ends 752, 755
  - flush 752, 756
  - hex 752, 757
  - oct 752, 758
  - resetiosflags 752, 759
  - setbase 752, 760
  - setfill 752, 761
  - setiosflags 752, 762
  - setprecision 752, 763
  - setw 752, 764
  - setwidth 752, 765
  - ws 752, 766
- match, member function
  - String 886, 893

## N

- nocreate, member enumeration
  - ios 694
- noreplace, member enumeration
  - ios 694
- norm, related function
  - Complex 21, 43
- not\_empty
  - exception 74, 93, 116, 120, 142, 164, 168, 245, 248, 269, 273, 297, 301, 432, 447, 467, 471, 490, 509, 513, 532, 545, 548, 576, 589, 592, 620
- not\_empty, member enumeration
  - WCEXcept 74
- not\_unique
  - exception 75, 129, 176, 478, 519
- not\_unique, member enumeration
  - WCEXcept 74
- num, related function

Complex 41

## O

- occurrencesOf, member function
  - WCPtrHashSet<Type> 112, 131
  - WCPtrHashTable<Type> 112, 131
  - WCPtrOrderedVector<Type> 541, 561
  - WCPtrSkipList<Type> 463, 480
  - WCPtrSkipListSet<Type> 463, 480
  - WCPtrSortedVector<Type> 541, 561
  - WCValHashSet<Type> 160, 178
  - WCValHashTable<Type> 160, 178
  - WCValOrderedVector<Type> 586, 604
  - WCValSkipList<Type> 505, 521
  - WCValSkipListSet<Type> 505, 521
  - WCValSortedVector<Type> 586, 604
- oct, manipulator 752, 758
- oct, member enumeration
  - ios 683
- ofstream 653, 774
- ofstream::ofstream 767-771
- ofstream::open 767, 773
- ofstream::~ofstream 767, 772
- open, member function
  - filebuf 629, 638
  - fstream 646, 652
  - fstreambase 653, 663
  - ifstream 666, 672
  - ofstream 767, 773
- openmode, member enumeration
  - ios 673, 694
- openprot, member data
  - filebuf 639
- openprot, member function
  - filebuf 628
- operator !, member function
  - ios 674, 696
  - String 886, 894
- operator !=, related function

- Complex 21, 44
- String 887, 895
- operator (), member function
  - String 886, 896-897
  - WCIsvConstDListIter<Type> 319, 328
  - WCIsvConstSListIter<Type> 319, 328
  - WCIsvDListIter<Type> 336, 347
  - WCIsvSListIter<Type> 336, 347
  - WCPtrConstDListIter<Type> 355, 364
  - WCPtrConstSListIter<Type> 355, 364
  - WCPtrDListIter<Type> 372, 383
  - WCPtrHashDictIter<Key, Value> 186, 192
  - WCPtrHashSetIter<Type> 208, 217
  - WCPtrHashTableIter<Type> 208, 217
  - WCPtrSListIter<Type> 372, 383
  - WCValConstDListIter<Type> 391, 400
  - WCValConstSListIter<Type> 391, 400
  - WCValDListIter<Type> 408, 420
  - WCValHashDictIter<Key, Value> 197, 203
  - WCValHashSetIter<Type> 221, 230
  - WCValHashTableIter<Type> 221, 230
  - WCValSListIter<Type> 408, 420
- operator \*, related function
  - Complex 21, 45
- operator \*=, member function
  - Complex 20, 46
- operator ++, member function
  - WCIsvConstDListIter<Type> 319, 329
  - WCIsvConstSListIter<Type> 319, 329
  - WCIsvDListIter<Type> 336, 348
  - WCIsvSListIter<Type> 336, 348
  - WCPtrConstDListIter<Type> 355, 365
  - WCPtrConstSListIter<Type> 355, 365
  - WCPtrDListIter<Type> 372, 384
  - WCPtrHashDictIter<Key, Value> 186, 193
  - WCPtrHashSetIter<Type> 208, 218
  - WCPtrHashTableIter<Type> 208, 218
  - WCPtrSListIter<Type> 372, 384
  - WCValConstDListIter<Type> 391, 401
  - WCValConstSListIter<Type> 391, 401
  - WCValDListIter<Type> 408, 421
  - WCValHashDictIter<Key, Value> 197, 204
  - WCValHashSetIter<Type> 221, 231
  - WCValHashTableIter<Type> 221, 231
- WCValSListIter<Type> 408, 421
- operator +, member function
  - Complex 20, 47
- operator +, related function
  - Complex 20-21, 48
  - String 887, 898
- operator +=, member function
  - Complex 20, 49
  - String 886, 899
  - WCIsvConstDListIter<Type> 319, 330
  - WCIsvConstSListIter<Type> 319, 330
  - WCIsvDListIter<Type> 336, 349
  - WCIsvSListIter<Type> 336, 349
  - WCPtrConstDListIter<Type> 355, 366
  - WCPtrConstSListIter<Type> 355, 366
  - WCPtrDListIter<Type> 372, 385
  - WCPtrSListIter<Type> 372, 385
  - WCValConstDListIter<Type> 391, 402
  - WCValConstSListIter<Type> 391, 402
  - WCValDListIter<Type> 408, 422
  - WCValSListIter<Type> 408, 422
- operator -, member function
  - Complex 20, 50
- operator -, related function
  - Complex 21, 51
- operator --, member function
  - WCIsvConstDListIter<Type> 319, 331
  - WCIsvConstSListIter<Type> 319, 331
  - WCIsvDListIter<Type> 335-336, 350
  - WCIsvSListIter<Type> 335-336, 350
  - WCPtrConstDListIter<Type> 355, 367
  - WCPtrConstSListIter<Type> 355, 367
  - WCPtrDListIter<Type> 371-372, 386
  - WCPtrSListIter<Type> 371-372, 386
  - WCValConstDListIter<Type> 391, 403
  - WCValConstSListIter<Type> 391, 403
  - WCValDListIter<Type> 407-408, 423
  - WCValSListIter<Type> 407-408, 423
- operator -=, member function
  - Complex 20, 52
  - WCIsvConstDListIter<Type> 319, 332
  - WCIsvConstSListIter<Type> 319, 332
  - WCIsvDListIter<Type> 335-336, 351
  - WCIsvSListIter<Type> 335-336, 351

- WCPtrConstDListIter<Type> 355, 368
- WCPtrConstSListIter<Type> 355, 368
- WCPtrDListIter<Type> 371-372, 387
- WCPtrSListIter<Type> 371-372, 387
- WCValConstDListIter<Type> 391, 404
- WCValConstSListIter<Type> 391, 404
- WCValDListIter<Type> 407-408, 424
- WCValSListIter<Type> 407-408, 424
- operator /, related function
  - Complex 21, 53
- operator /=, member function
  - Complex 20, 54
- operator <, related function
  - String 887, 900
- operator <<, member function
  - ostream 775, 777-779, 781-784
- operator <<, related function
  - Complex 20, 55
  - String 887, 901
- operator <=, related function
  - String 887, 902
- operator =, member function
  - Complex 20, 56
  - iostream 710, 715-716
  - istream 718, 733-734
  - ostream 775, 785-786
  - String 886, 903
  - WCIsvDList<Type> 240, 262
  - WCIsvSList<Type> 240, 262
  - WCPtrDList<Type> 265, 286
  - WCPtrHashDict<Key, Value> 89, 107
  - WCPtrHashSet<Type> 112, 132
  - WCPtrHashTable<Type> 112, 132
  - WCPtrOrderedVector<Type> 542, 563
  - WCPtrSkipList<Type> 463, 481
  - WCPtrSkipListDict<Key, Value> 443, 459
  - WCPtrSkipListSet<Type> 463, 481
  - WCPtrSList<Type> 265, 286
  - WCPtrSortedVector<Type> 542, 563
  - WCPtrVector<Type> 572, 581
  - WCValDList<Type> 292, 314
  - WCValHashDict<Key, Value> 138, 155
  - WCValHashSet<Type> 160, 179
  - WCValHashTable<Type> 160, 179
  - WCValOrderedVector<Type> 586, 606
  - WCValSkipList<Type> 505, 522
  - WCValSkipListDict<Key, Value> 486, 501
  - WCValSkipListSet<Type> 505, 522
  - WCValSList<Type> 292, 314
  - WCValSortedVector<Type> 586, 606
  - WCValVector<Type> 616, 624
- operator ==, member function
  - WCIsvDList<Type> 241, 263
  - WCIsvSList<Type> 241, 263
  - WCPtrDList<Type> 265, 287
  - WCPtrHashDict<Key, Value> 89, 108
  - WCPtrHashSet<Type> 112, 133
  - WCPtrHashTable<Type> 112, 133
  - WCPtrOrderedVector<Type> 542, 564
  - WCPtrSkipList<Type> 463, 482
  - WCPtrSkipListDict<Key, Value> 443, 460
  - WCPtrSkipListSet<Type> 463, 482
  - WCPtrSList<Type> 265, 287
  - WCPtrSortedVector<Type> 542, 564
  - WCPtrVector<Type> 572, 582
  - WCValDList<Type> 292, 315
  - WCValHashDict<Key, Value> 138, 156
  - WCValHashSet<Type> 160, 180
  - WCValHashTable<Type> 160, 180
  - WCValOrderedVector<Type> 586, 607
  - WCValSkipList<Type> 505, 523
  - WCValSkipListDict<Key, Value> 486, 502
  - WCValSkipListSet<Type> 505, 523
  - WCValSList<Type> 292, 315
  - WCValSortedVector<Type> 586, 607
  - WCValVector<Type> 616, 625
- operator ==, related function
  - Complex 21, 57
  - String 887, 904
- operator >, related function
  - String 887, 905
- operator >=, related function
  - String 887, 906
- operator >>, member function
  - istream 718, 735-740
- operator >>, related function
  - Complex 20, 58
  - String 887, 907



- operator [], member function
    - String 886, 908
    - WCPtrHashDict<Key,Value> 89, 105-106
    - WCPtrOrderedVector<Type> 542, 562
    - WCPtrSkipListDict<Key,Value> 443, 457-458
    - WCPtrSortedVector<Type> 542, 562
    - WCPtrVector<Type> 572, 580
    - WCValHashDict<Key,Value> 138, 153-154
    - WCValOrderedVector<Type> 586, 605
    - WCValSkipListDict<Key,Value> 486, 499-500
    - WCValSortedVector<Type> 586, 605
    - WCValVector<Type> 616, 623
  - operator char const \*, member function
    - String 886, 910
  - operator char, member function
    - String 886, 909
  - operator void \*, member function
    - ios 674, 697
  - opfx, member function
    - ostream 774, 787
  - osfx, member function
    - ostream 774, 788
  - ostream 673, 710, 767, 798
  - ostream output 15
  - ostream::flush 774, 776
  - ostream::operator << 775, 777-779, 781-784
  - ostream::operator = 775, 785-786
  - ostream::opfx 774, 787
  - ostream::osfx 774, 788
  - ostream::ostream 774, 789-791
  - ostream::put 774, 793
  - ostream::seekp 774-775, 794-795
  - ostream::tellp 775, 796
  - ostream::write 775, 797
  - ostream::~ostream 774, 792
  - ostrstream 774, 862
  - ostrstream::ostrstream 798-800
  - ostrstream::pcount 798, 802
  - ostrstream::str 798, 803
  - ostrstream::~ostrstream 798, 801
  - out, member enumeration
    - ios 694
  - out\_of\_memory 379, 382, 416, 419
    - exception 74, 90, 92, 103, 105, 107, 110, 113, 115, 117, 119, 129, 132, 136, 139, 141, 151, 153, 155, 158, 161, 163, 165, 167, 176, 179, 183, 268, 272, 274, 284, 286, 296, 300, 302, 312, 314, 437, 444, 446, 455, 457, 459, 464, 466, 468, 470, 478, 481, 487, 489, 497, 499, 501, 506, 508, 510, 512, 519, 522, 537, 544, 547, 549, 557-558, 563, 565, 571, 575, 580-581, 583, 588, 591, 593, 600-601, 606, 608, 614, 619, 623-624, 626
  - out\_of\_memory, member enumeration
    - WCEXcept 74
  - out\_waiting, member function
    - streambuf 813, 829
  - overflow, member function
    - filebuf 629, 640
    - stdiobuf 804-805
    - streambuf 813, 830
    - strstreambuf 868, 872
- P
- pbackfail, member function
    - filebuf 629, 641
    - streambuf 813, 831
  - pbase, member function
    - streambuf 813, 832
  - pbump, member function
    - streambuf 813, 833
  - pcount, member function
    - ostrstream 798, 802
  - peek, member function
    - istream 718, 741
  - pointer
    - lists 235
  - polar, related function
    - Complex 21, 59
  - pop, member function
    - WCStack<Type,FType> 528, 536

pow, related function  
  Complex 21, 60  
pptr, member function  
  streambuf 813, 834  
precision, member function  
  ios 674, 698  
predefined objects 11  
prepend, member function  
  WCPtrOrderedVector<Type> 542, 565  
  WCPtrSortedVector<Type> 542, 565  
  WCValOrderedVector<Type> 586, 608  
  WCValSortedVector<Type> 586, 608  
push, member function  
  WCStack<Type,FType> 528, 537  
put area 811  
put pointer 834  
put, member function  
  ostream 774, 793  
put\_at, member function  
  String 886, 911  
putback, member function  
  istream 718, 742  
pword, member function  
  ios 674, 699

## R

rdbuf, member function  
  fstreambase 653, 664  
  ios 673, 700  
  strstreambase 862-863  
rdstate, member function  
  ios 673, 701  
read, member function  
  istream 718, 743  
real, member function  
  Complex 20, 61  
real, related function  
  Complex 21, 62  
remove, member function

  WCPtrHashDict<Key,Value> 89, 109  
  WCPtrHashSet<Type> 112, 134  
  WCPtrHashTable<Type> 112, 134  
  WCPtrOrderedVector<Type> 541, 566  
  WCPtrSkipList<Type> 463, 483  
  WCPtrSkipListDict<Key,Value> 443, 461  
  WCPtrSkipListSet<Type> 463, 483  
  WCPtrSortedVector<Type> 541, 566  
  WCValHashDict<Key,Value> 138, 157  
  WCValHashSet<Type> 160, 181  
  WCValHashTable<Type> 160, 181  
  WCValOrderedVector<Type> 586, 609  
  WCValSkipList<Type> 505, 524  
  WCValSkipListDict<Key,Value> 486, 503  
  WCValSkipListSet<Type> 505, 524  
  WCValSortedVector<Type> 586, 609  
removeAll, member function  
  WCPtrHashSet<Type> 112, 135  
  WCPtrHashTable<Type> 112, 135  
  WCPtrOrderedVector<Type> 541, 567  
  WCPtrSkipList<Type> 463, 484  
  WCPtrSkipListSet<Type> 463, 484  
  WCPtrSortedVector<Type> 541, 567  
  WCValHashSet<Type> 160, 182  
  WCValHashTable<Type> 160, 182  
  WCValOrderedVector<Type> 586, 610  
  WCValSkipList<Type> 505, 525  
  WCValSkipListSet<Type> 505, 525  
  WCValSortedVector<Type> 586, 610  
removeAt, member function  
  WCPtrOrderedVector<Type> 541, 568  
  WCPtrSortedVector<Type> 541, 568  
  WCValOrderedVector<Type> 586, 611  
  WCValSortedVector<Type> 586, 611  
removeFirst, member function  
  WCPtrOrderedVector<Type> 541, 569  
  WCPtrSortedVector<Type> 541, 569  
  WCValOrderedVector<Type> 586, 612  
  WCValSortedVector<Type> 586, 612  
removeLast, member function  
  WCPtrOrderedVector<Type> 541, 570  
  WCPtrSortedVector<Type> 541, 570  
  WCValOrderedVector<Type> 586, 613  
  WCValSortedVector<Type> 586, 613

- reserve area 811  
 reset, member function  
   WCIsvConstDListIter<Type> 318-319, 333-334  
   WCIsvConstSListIter<Type> 318-319, 333-334  
   WCIsvDListIter<Type> 335, 352-353  
   WCIsvSListIter<Type> 335, 352-353  
   WCPtrConstDListIter<Type> 354-355, 369-370  
   WCPtrConstSListIter<Type> 354-355, 369-370  
   WCPtrDListIter<Type> 371, 388-389  
   WCPtrHashDictIter<Key,Value> 186, 194-195  
   WCPtrHashSetIter<Type> 208, 219-220  
   WCPtrHashTableIter<Type> 208, 219-220  
   WCPtrSListIter<Type> 371, 388-389  
   WCValConstDListIter<Type> 390-391, 405-406  
   WCValConstSListIter<Type> 390-391, 405-406  
   WCValDListIter<Type> 407, 425-426  
   WCValHashDictIter<Key,Value> 197, 205-206  
   WCValHashSetIter<Type> 221, 232-233  
   WCValHashTableIter<Type> 221, 232-233  
   WCValSListIter<Type> 407, 425-426  
 resetiosflags, manipulator 752, 759  
 resize, member function  
   WCPtrHashDict<Key,Value> 89, 110  
   WCPtrHashSet<Type> 112, 136  
   WCPtrHashTable<Type> 112, 136  
   WCPtrOrderedVector<Type> 541, 571  
   WCPtrSortedVector<Type> 541, 571  
   WCPtrVector<Type> 572, 583  
   WCValHashDict<Key,Value> 138, 158  
   WCValHashSet<Type> 160, 183  
   WCValHashTable<Type> 160, 183  
   WCValOrderedVector<Type> 586, 614  
   WCValSortedVector<Type> 586, 614  
   WCValVector<Type> 616, 626  
 resize\_required  
   exception 75, 540, 543, 546, 549, 557-558, 565, 580, 584, 587, 590, 593, 600-601, 608, 623  
   resize\_required, member enumeration  
     WCExcept 74  
   right, member enumeration  
     ios 683

<b>S</b>
----------

- sbumpc, member function  
   streambuf 813, 835  
 scientific, member enumeration  
   ios 683  
 seekdir, member enumeration  
   ios 673, 702  
 seekg, member function  
   istream 718, 744-745  
 seekoff, member function  
   filebuf 629, 642  
   streambuf 814, 836  
   strstreambuf 868, 873  
 seekp, member function  
   ostream 774-775, 794-795  
 seekpos, member function  
   streambuf 814, 837  
 setb, member function  
   streambuf 813, 838  
 setbase, manipulator 752, 760  
 setbuf, member function  
   filebuf 629, 643  
   fstreambase 653, 665  
   streambuf 814, 839  
   strstreambuf 868, 874  
 setf, member function  
   ios 674, 703  
 setfill, manipulator 752, 761  
 setg, member function  
   streambuf 813, 840  
 setiosflags, manipulator 752, 762

- setp, member function
  - streambuf 813, 841
- setprecision, manipulator 752, 763
- setstate, member function
  - ios 673, 704
- setw, manipulator 752, 764
- setWidth, manipulator 752, 765
- sgetc, member function
  - streambuf 813, 842
- sgetchar, member function
  - streambuf 813, 843
- sgetn, member function
  - streambuf 813, 844
- showbase, member enumeration
  - ios 683
- showpoint, member enumeration
  - ios 683
- showpos, member enumeration
  - ios 683
- sin, related function
  - Complex 21, 63
- sinh, related function
  - Complex 21, 64
- skipws, member enumeration
  - ios 683
- snxctc, member function
  - streambuf 813, 845
- speekc, member function
  - streambuf 813, 846
- sputbackc, member function
  - streambuf 813, 847
- sputc, member function
  - streambuf 813, 848
- sputn, member function
  - streambuf 813, 849
- sqrt, related function
  - Complex 21, 65
- stdio, member enumeration
  - ios 683
- stdiobuf 811
- stdiobuf::overflow 804-805
- stdiobuf::stdiobuf 804, 806-807
- stdiobuf::sync 804, 809
- stdiobuf::underflow 804, 810
- stdiobuf::~stdiobuf 804, 808
- stoss, member function
  - streambuf 813, 850
- str, member function
  - ostrstream 798, 803
  - strstream 857-858
  - strstreambuf 867, 875
- streambuf 628, 804, 867
- streambuf::allocate 813, 815
- streambuf::base 813, 816
- streambuf::blen 813, 817
- streambuf::dbp 813, 818
- streambuf::do\_sgetn 813, 819
- streambuf::do\_sputn 813, 820
- streambuf::doallocate 813, 821
- streambuf::eback 813, 822
- streambuf::ebuf 813, 823
- streambuf::egptr 813, 824
- streambuf::epptr 813, 825
- streambuf::gbump 813, 826
- streambuf::gptr 813, 827
- streambuf::in\_avail 813, 828
- streambuf::out\_waiting 813, 829
- streambuf::overflow 813, 830
- streambuf::pbackfail 813, 831
- streambuf::pbase 813, 832
- streambuf::pbump 813, 833
- streambuf::pptr 813, 834
- streambuf::sbumpc 813, 835
- streambuf::seekoff 814, 836
- streambuf::seekpos 814, 837
- streambuf::setb 813, 838
- streambuf::setbuf 814, 839
- streambuf::setg 813, 840
- streambuf::setp 813, 841
- streambuf::sgetc 813, 842
- streambuf::sgetchar 813, 843
- streambuf::sgetn 813, 844
- streambuf::snxctc 813, 845
- streambuf::speekc 813, 846
- streambuf::sputbackc 813, 847
- streambuf::sputc 813, 848
- streambuf::sputn 813, 849
- streambuf::stoss 813, 850

- streambuf::streambuf 813, 851-852
- streambuf::sync 814, 854
- streambuf::unbuffered 813, 855
- streambuf::underflow 814, 856
- streambuf::~streambuf 813, 853
- streamoff 9
- streampos 9
- String related functions
  - operator != 887, 895
  - operator + 887, 898
  - operator < 887, 900
  - operator << 887, 901
  - operator <= 887, 902
  - operator == 887, 904
  - operator > 887, 905
  - operator >= 887, 906
  - operator >> 887, 907
  - valid 887, 919
- String::alloc\_mult\_size 886, 888
- String::get\_at 886, 889
- String::index 886, 890
- String::length 886, 891
- String::lower 886, 892
- String::match 886, 893
- String::operator ! 886, 894
- String::operator () 886, 896-897
- String::operator += 886, 899
- String::operator = 886, 903
- String::operator [] 886, 908
- String::operator char 886, 909
- String::operator char const \* 886, 910
- String::put\_at 886, 911
- String::String 886, 912-916
- String::upper 886, 918
- String::valid 886, 920
- String::~String 886, 917
- strstream 710, 862
- strstream::str 857-858
- strstream::strstream 857, 859-860
- strstream::~strstream 857, 861
- strstreambase 748, 798, 857
- strstreambase::rdbuf 862-863
- strstreambase::strstreambase 862, 864-865
- strstreambase::~strstreambase 862, 866
- strstreambuf 811
- strstreambuf::alloc\_size\_increment 867, 869
- strstreambuf::doallocate 867, 870
- strstreambuf::freeze 867, 871
- strstreambuf::overflow 868, 872
- strstreambuf::seekoff 868, 873
- strstreambuf::setbuf 868, 874
- strstreambuf::str 867, 875
- strstreambuf::strstreambuf 867, 876-879
- strstreambuf::sync 868, 882
- strstreambuf::underflow 868, 883
- strstreambuf::~strstreambuf 867, 881
- sync, member function
  - filebuf 629, 644
  - istream 718, 746
  - stdiobuf 804, 809
  - streambuf 814, 854
  - strstreambuf 868, 882
- sync\_with\_stdio, member function
  - ios 674, 705

T

- tan, related function
  - Complex 22, 66
- tanh, related function
  - Complex 22, 67
- tellg, member function
  - istream 718, 747
- tellp, member function
  - ostream 775, 796
- text, member enumeration
  - ios 694
- tie, member function
  - ios 673, 706
- top, member function
  - WCString<Type,FType> 528, 538
- trunc, member enumeration
  - ios 694
- truncate, member enumeration

ios 694

## U

unbuffered, member function  
  streambuf 813, 855  
undef\_item 191, 196, 202, 207, 216, 229, 327,  
  345, 363, 381, 399, 418  
  exception 80  
undef\_item, member enumeration  
  WCIterExcept 80  
undef\_iter  
  exception 80, 190, 192-193, 201, 203-204,  
  215, 217-218, 228, 230-231, 326,  
  328-332, 343-344, 346-351, 362,  
  364-368, 379-380, 382-387, 398,  
  400-404, 416-417, 419-424  
undef\_iter, member enumeration  
  WCIterExcept 80  
underflow, member function  
  filebuf 629, 645  
  stdiobuf 804, 810  
  streambuf 814, 856  
  strstreambuf 868, 883  
undex\_iter  
  exception 185, 317  
unformatted input 13  
unformatted output 15  
unitbuf, member enumeration  
  ios 683  
unsetf, member function  
  ios 674, 707  
upper, member function  
  String 886, 918  
uppercase, member enumeration  
  ios 683

## V

valid, member function  
  String 886, 920  
valid, related function  
  String 887, 919  
value  
  lists 235  
value, member function  
  WCPtrHashDictIter<Key, Value> 186, 196  
  WCValHashDictIter<Key, Value> 197, 207

## W

wc\_state, member enumeration  
  WCEexcept 70, 74  
WCDLink 288  
WCDLink::WCDLink 237-238  
WCDLink::~~WCDLink 237, 239  
WCEexcept::all\_fine 74  
WCEexcept::check\_all 74  
WCEexcept::check\_none 74  
WCEexcept::empty\_container 74  
WCEexcept::exceptions 70, 73  
WCEexcept::index\_range 74  
WCEexcept::not\_empty 74  
WCEexcept::not\_unique 74  
WCEexcept::out\_of\_memory 74  
WCEexcept::resize\_required 74  
WCEexcept::wc\_state 70, 74  
WCEexcept::WCEexcept 70-71  
WCEexcept::zero\_buckets 74  
WCEexcept::~~WCEexcept 70, 72  
WCIsvConstDListIter, member function  
  WCIsvConstDListIter<Type> 323-324  
  WCIsvConstSListIter<Type> 323-324  
WCIsvConstDListIter<Type>::append 318



WCIsvDListIter<Type>::WCIsvDListIter  
     340-341  
 WCIsvDListIter<Type>::WCIsvSListIter  
     337-338  
 WCIsvDListIter<Type>::~~WCIsvDListIter 342  
 WCIsvDListIter<Type>::~~WCIsvSListIter 339  
 WCIsvSList, member function  
     WCIsvDList<Type> 240, 243, 245  
     WCIsvSList<Type> 240, 243, 245  
 WCIsvSList<Type>::append 241, 249  
 WCIsvSList<Type>::clear 241, 250  
 WCIsvSList<Type>::clearAndDestroy 241, 251  
 WCIsvSList<Type>::contains 241, 252  
 WCIsvSList<Type>::entries 241, 253  
 WCIsvSList<Type>::find 241, 254  
 WCIsvSList<Type>::findLast 241, 255  
 WCIsvSList<Type>::forAll 241, 256  
 WCIsvSList<Type>::get 241, 257  
 WCIsvSList<Type>::index 241, 258-259  
 WCIsvSList<Type>::insert 241, 260  
 WCIsvSList<Type>::isEmpty 241, 261  
 WCIsvSList<Type>::operator = 240, 262  
 WCIsvSList<Type>::operator == 241, 263  
 WCIsvSList<Type>::WCIsvDList 240, 244,  
     246-248  
 WCIsvSList<Type>::WCIsvSList 240, 243, 245  
 WCIsvSList<Type>::WCIsvSList<Type>  
     240-241  
 WCIsvSList<Type>::~~WCIsvSList<Type>  
     240-241  
 WCIsvSListIter, member function  
     WCIsvDListIter<Type> 337-338  
     WCIsvSListIter<Type> 337-338  
 WCIsvSListIter<Type>::append 335, 343  
 WCIsvSListIter<Type>::container 335, 344  
 WCIsvSListIter<Type>::current 335, 345  
 WCIsvSListIter<Type>::insert 335-336, 346  
 WCIsvSListIter<Type>::operator () 336, 347  
 WCIsvSListIter<Type>::operator ++ 336, 348  
 WCIsvSListIter<Type>::operator += 336, 349  
 WCIsvSListIter<Type>::operator -- 335-336, 350  
 WCIsvSListIter<Type>::operator -= 335-336,  
     351  
 WCIsvSListIter<Type>::reset 335, 352-353  
 WCIsvSListIter<Type>::WCIsvDListIter  
     340-341  
 WCIsvSListIter<Type>::WCIsvSListIter 337-338  
 WCIsvSListIter<Type>::WCIsvSListIter<Type>  
     335  
 WCIsvSListIter<Type>::~~WCIsvDListIter 342  
 WCIsvSListIter<Type>::~~WCIsvSListIter 339  
 WCIsvSListIter<Type>::~~WCIsvSListIter<Type>  
     335  
 w citer\_state, member enumeration  
     WCIterExcept 76, 80  
 WCIterExcept::all\_fine 80  
 WCIterExcept::check\_all 80  
 WCIterExcept::check\_none 80  
 WCIterExcept::exceptions 76, 79  
 WCIterExcept::iter\_range 80  
 WCIterExcept::undef\_item 80  
 WCIterExcept::undef\_iter 80  
 WCIterExcept::w citer\_state 76, 80  
 WCIterExcept::WCIterExcept 76-77  
 WCIterExcept::~~WCIterExcept 76, 78  
 WCListExcept  
     class 70  
 WCPtrConstDListIter, member function  
     WCPtrConstDListIter<Type> 359-360  
     WCPtrConstSListIter<Type> 359-360  
 WCPtrConstDListIter<Type>::append 354  
 WCPtrConstDListIter<Type>::container 354, 362  
 WCPtrConstDListIter<Type>::current 354, 363  
 WCPtrConstDListIter<Type>::insert 354  
 WCPtrConstDListIter<Type>::operator () 355,  
     364  
 WCPtrConstDListIter<Type>::operator ++ 355,  
     365  
 WCPtrConstDListIter<Type>::operator += 355,  
     366  
 WCPtrConstDListIter<Type>::operator -- 355,  
     367  
 WCPtrConstDListIter<Type>::operator -= 355,  
     368  
 WCPtrConstDListIter<Type>::reset 354-355,  
     369-370  
 WCPtrConstDListIter<Type>::WCPtrConstDListI  
     ter 359-360



- WPtrConstDListIter<Type>::WPtrConstSListIter 356-357
- WPtrConstDListIter<Type>::~~WPtrConstDListIter 361
- WPtrConstDListIter<Type>::~~WPtrConstSListIter 358
- WPtrConstSListIter, member function
  - WPtrConstDListIter<Type> 356-357
  - WPtrConstSListIter<Type> 356-357
- WPtrConstSListIter<Type>::append 354
- WPtrConstSListIter<Type>::container 354, 362
- WPtrConstSListIter<Type>::current 354, 363
- WPtrConstSListIter<Type>::insert 354
- WPtrConstSListIter<Type>::operator () 355, 364
- WPtrConstSListIter<Type>::operator ++ 355, 365
- WPtrConstSListIter<Type>::operator += 355, 366
- WPtrConstSListIter<Type>::operator -- 355, 367
- WPtrConstSListIter<Type>::operator -= 355, 368
- WPtrConstSListIter<Type>::reset 354-355, 369-370
- WPtrConstSListIter<Type>::WPtrConstDListIter 359-360
- WPtrConstSListIter<Type>::WPtrConstSListIter 356-357
- WPtrConstSListIter<Type>::WPtrConstSListIter<Type> 354
- WPtrConstSListIter<Type>::~~WPtrConstDListIter 361
- WPtrConstSListIter<Type>::~~WPtrConstSListIter 358
- WPtrConstSListIter<Type>::~~WPtrConstSListIter<Type> 354
- WPtrDList, member function
  - WPtrDList<Type> 268, 270-273
  - WPtrSList<Type> 268, 270-273
- WPtrDList<Type>::append 264, 274
- WPtrDList<Type>::clear 264, 275
- WPtrDList<Type>::clearAndDestroy 264, 276
- WPtrDList<Type>::contains 264, 277
- WPtrDList<Type>::entries 264, 278
- WPtrDList<Type>::find 264, 279
- WPtrDList<Type>::findLast 264, 280
- WPtrDList<Type>::forAll 264, 281
- WPtrDList<Type>::get 264, 282
- WPtrDList<Type>::index 264, 283
- WPtrDList<Type>::insert 264, 284
- WPtrDList<Type>::isEmpty 264, 285
- WPtrDList<Type>::operator = 265, 286
- WPtrDList<Type>::operator == 265, 287
- WPtrDList<Type>::WPtrDList 268, 270-273
- WPtrDList<Type>::WPtrSList 266-267, 269
- WPtrDListIter, member function
  - WPtrDListIter<Type> 376-377
  - WPtrSListIter<Type> 376-377
- WPtrDListIter<Type>::append 371, 379
- WPtrDListIter<Type>::container 371, 380
- WPtrDListIter<Type>::current 371, 381
- WPtrDListIter<Type>::insert 371-372, 382
- WPtrDListIter<Type>::operator () 372, 383
- WPtrDListIter<Type>::operator ++ 372, 384
- WPtrDListIter<Type>::operator += 372, 385
- WPtrDListIter<Type>::operator -- 371-372, 386
- WPtrDListIter<Type>::operator -= 371-372, 387
- WPtrDListIter<Type>::reset 371, 388-389
- WPtrDListIter<Type>::WPtrDListIter 376-377
- WPtrDListIter<Type>::WPtrSListIter 373-374
- WPtrDListIter<Type>::~~WPtrDListIter 378
- WPtrDListIter<Type>::~~WPtrSListIter 375
- WPtrHashDict, member function
  - WPtrHashDict<Key, Value> 89
- WPtrHashDict<Key, Value>::bitHash 89, 94
- WPtrHashDict<Key, Value>::buckets 89, 95
- WPtrHashDict<Key, Value>::clear 89, 96
- WPtrHashDict<Key, Value>::clearAndDestroy 89, 97
- WPtrHashDict<Key, Value>::contains 89, 98
- WPtrHashDict<Key, Value>::entries 89, 99
- WPtrHashDict<Key, Value>::find 89, 100
- WPtrHashDict<Key, Value>::findKeyAndValue 89, 101
- WPtrHashDict<Key, Value>::forall 89, 102

- WCPtrHashDict<Key, Value>::insert 89, 103
- WCPtrHashDict<Key, Value>::isEmpty 89, 104
- WCPtrHashDict<Key, Value>::operator = 89, 107
- WCPtrHashDict<Key, Value>::operator == 89, 108
- WCPtrHashDict<Key, Value>::operator [] 89, 105-106
- WCPtrHashDict<Key, Value>::remove 89, 109
- WCPtrHashDict<Key, Value>::resize 89, 110
- WCPtrHashDict<Key, Value>::WCPtrHashDict 89
- WCPtrHashDict<Key, Value>::WCPtrHashDict<Key, Value> 90-92
- WCPtrHashDict<Key, Value>::~~WCPtrHashDict 89
- WCPtrHashDict<Key, Value>::~~WCPtrHashDict<Key, Value> 93
- WCPtrHashDictIter, member function
  - WCPtrHashDictIter<Key, Value> 187-188
- WCPtrHashDictIter<Key, Value>::container 186, 190
- WCPtrHashDictIter<Key, Value>::key 186, 191
- WCPtrHashDictIter<Key, Value>::operator () 186, 192
- WCPtrHashDictIter<Key, Value>::operator ++ 186, 193
- WCPtrHashDictIter<Key, Value>::reset 186, 194-195
- WCPtrHashDictIter<Key, Value>::value 186, 196
- WCPtrHashDictIter<Key, Value>::WCPtrHashDictIter 187-188
- WCPtrHashDictIter<Key, Value>::WCPtrHashDictIter<Key, Value> 186
- WCPtrHashDictIter<Key, Value>::~~WCPtrHashDictIter 189
- WCPtrHashDictIter<Key, Value>::~~WCPtrHashDictIter<Key, Value> 186
- WCPtrHashSet, member function
  - WCPtrHashSet<Type> 112
  - WCPtrHashTable<Type> 112
- WCPtrHashSet<Type>::bitHash 112, 121
- WCPtrHashSet<Type>::buckets 112, 122
- WCPtrHashSet<Type>::clear 112, 123
- WCPtrHashSet<Type>::clearAndDestroy 112, 124
- WCPtrHashSet<Type>::contains 112, 125
- WCPtrHashSet<Type>::entries 112, 126
- WCPtrHashSet<Type>::find 112, 127
- WCPtrHashSet<Type>::forall 112, 128
- WCPtrHashSet<Type>::insert 112, 129
- WCPtrHashSet<Type>::isEmpty 112, 130
- WCPtrHashSet<Type>::occurrencesOf 112, 131
- WCPtrHashSet<Type>::operator = 112, 132
- WCPtrHashSet<Type>::operator == 112, 133
- WCPtrHashSet<Type>::remove 112, 134
- WCPtrHashSet<Type>::removeAll 112, 135
- WCPtrHashSet<Type>::resize 112, 136
- WCPtrHashSet<Type>::WCPtrHashSet 112
- WCPtrHashSet<Type>::WCPtrHashTable 112
- WCPtrHashSet<Type>::~~WCPtrHashSet 112
- WCPtrHashSet<Type>::~~WCPtrHashTable 112
- WCPtrHashSetIter, member function
  - WCPtrHashSetIter<Type> 209-210
  - WCPtrHashTableIter<Type> 209-210
- WCPtrHashSetIter<Type>::container 208, 215
- WCPtrHashSetIter<Type>::current 208, 216
- WCPtrHashSetIter<Type>::operator () 208, 217
- WCPtrHashSetIter<Type>::operator ++ 208, 218
- WCPtrHashSetIter<Type>::reset 208, 219-220
- WCPtrHashSetIter<Type>::WCPtrHashSetIter 209-210
- WCPtrHashSetIter<Type>::WCPtrHashSetIter<Type> 208
- WCPtrHashSetIter<Type>::WCPtrHashTableIter 212-213
- WCPtrHashSetIter<Type>::~~WCPtrHashSetIter 211
- WCPtrHashSetIter<Type>::~~WCPtrHashSetIter<Type> 208
- WCPtrHashSetIter<Type>::~~WCPtrHashTableIter 214
- WCPtrHashTable, member function
  - WCPtrHashSet<Type> 112
  - WCPtrHashTable<Type> 112
- WCPtrHashTable<Type>::bitHash 112, 121
- WCPtrHashTable<Type>::buckets 112, 122
- WCPtrHashTable<Type>::clear 112, 123

- WCPtrHashTable<Type>::clearAndDestroy 112, 124
- WCPtrHashTable<Type>::contains 112, 125
- WCPtrHashTable<Type>::entries 112, 126
- WCPtrHashTable<Type>::find 112, 127
- WCPtrHashTable<Type>::forall 112, 128
- WCPtrHashTable<Type>::insert 112, 129
- WCPtrHashTable<Type>::isEmpty 112, 130
- WCPtrHashTable<Type>::occurrencesOf 112, 131
- WCPtrHashTable<Type>::operator = 112, 132
- WCPtrHashTable<Type>::operator == 112, 133
- WCPtrHashTable<Type>::remove 112, 134
- WCPtrHashTable<Type>::removeAll 112, 135
- WCPtrHashTable<Type>::resize 112, 136
- WCPtrHashTable<Type>::WCPtrHashSet 112
- WCPtrHashTable<Type>::WCPtrHashTable 112
- WCPtrHashTable<Type>::WCPtrHashTable<Type> e> 113-115, 117-119
- WCPtrHashTable<Type>::~WCPtrHashSet 112
- WCPtrHashTable<Type>::~WCPtrHashTable 112
- WCPtrHashTable<Type>::~WCPtrHashTable<Type> 116, 120
- WCPtrHashTableIter, member function
  - WCPtrHashSetIter<Type> 212-213
  - WCPtrHashTableIter<Type> 212-213
- WCPtrHashTableIter<Type>::container 208, 215
- WCPtrHashTableIter<Type>::current 208, 216
- WCPtrHashTableIter<Type>::operator () 208, 217
- WCPtrHashTableIter<Type>::operator ++ 208, 218
- WCPtrHashTableIter<Type>::reset 208, 219-220
- WCPtrHashTableIter<Type>::WCPtrHashSetIter 209-210
- WCPtrHashTableIter<Type>::WCPtrHashTableIter 212-213
- WCPtrHashTableIter<Type>::~WCPtrHashSetIter 211
- WCPtrHashTableIter<Type>::~WCPtrHashTableIter 214
- WCPtrOrderedVector, member function
  - WCPtrOrderedVector<Type> 541
  - WCPtrSortedVector<Type> 541
  - WCPtrOrderedVector<Type>::append 542, 549
  - WCPtrOrderedVector<Type>::clear 541, 550
  - WCPtrOrderedVector<Type>::clearAndDestroy 541, 551
  - WCPtrOrderedVector<Type>::contains 541, 552
  - WCPtrOrderedVector<Type>::entries 541, 553
  - WCPtrOrderedVector<Type>::find 541, 554
  - WCPtrOrderedVector<Type>::first 541, 555
  - WCPtrOrderedVector<Type>::index 541, 556
  - WCPtrOrderedVector<Type>::insert 541, 557
  - WCPtrOrderedVector<Type>::insertAt 542, 558
  - WCPtrOrderedVector<Type>::isEmpty 541, 559
  - WCPtrOrderedVector<Type>::last 541, 560
  - WCPtrOrderedVector<Type>::occurrencesOf 541, 561
  - WCPtrOrderedVector<Type>::operator = 542, 563
  - WCPtrOrderedVector<Type>::operator == 542, 564
  - WCPtrOrderedVector<Type>::operator [] 542, 562
  - WCPtrOrderedVector<Type>::prepend 542, 565
  - WCPtrOrderedVector<Type>::remove 541, 566
  - WCPtrOrderedVector<Type>::removeAll 541, 567
  - WCPtrOrderedVector<Type>::removeAt 541, 568
  - WCPtrOrderedVector<Type>::removeFirst 541, 569
  - WCPtrOrderedVector<Type>::removeLast 541, 570
  - WCPtrOrderedVector<Type>::resize 541, 571
  - WCPtrOrderedVector<Type>::WCPtrOrderedVector 541
  - WCPtrOrderedVector<Type>::WCPtrSortedVector 541
  - WCPtrOrderedVector<Type>::~WCPtrOrderedVector 541
  - WCPtrOrderedVector<Type>::~WCPtrSortedVector 541
  - WCPtrSkipList, member function
    - WCPtrSkipList<Type> 462-463
    - WCPtrSkipListSet<Type> 462-463

- WCPtrSkipList<Type>::clear 463, 472
- WCPtrSkipList<Type>::clearAndDestroy 463, 473
- WCPtrSkipList<Type>::contains 463, 474
- WCPtrSkipList<Type>::entries 463, 475
- WCPtrSkipList<Type>::find 463, 476
- WCPtrSkipList<Type>::forall 463, 477
- WCPtrSkipList<Type>::insert 463, 478
- WCPtrSkipList<Type>::isEmpty 463, 479
- WCPtrSkipList<Type>::occurrencesOf 463, 480
- WCPtrSkipList<Type>::operator = 463, 481
- WCPtrSkipList<Type>::operator == 463, 482
- WCPtrSkipList<Type>::remove 463, 483
- WCPtrSkipList<Type>::removeAll 463, 484
- WCPtrSkipList<Type>::WCPtrSkipList 462-463
- WCPtrSkipList<Type>::WCPtrSkipList<Type> 464-466, 468-470
- WCPtrSkipList<Type>::WCPtrSkipListSet 463
- WCPtrSkipList<Type>::~~WCPtrSkipList 463
- WCPtrSkipList<Type>::~~WCPtrSkipList<Type> 467, 471
- WCPtrSkipList<Type>::~~WCPtrSkipListSet 463
- WCPtrSkipListDict, member function
  - WCPtrSkipListDict<Key, Value> 442-443
- WCPtrSkipListDict<Key, Value>::clear 443, 448
- WCPtrSkipListDict<Key, Value>::clearAndDestroy 443, 449
- WCPtrSkipListDict<Key, Value>::contains 443, 450
- WCPtrSkipListDict<Key, Value>::entries 443, 451
- WCPtrSkipListDict<Key, Value>::find 443, 452
- WCPtrSkipListDict<Key, Value>::findKeyAndValue 443, 453
- WCPtrSkipListDict<Key, Value>::forall 443, 454
- WCPtrSkipListDict<Key, Value>::insert 443, 455
- WCPtrSkipListDict<Key, Value>::isEmpty 443, 456
- WCPtrSkipListDict<Key, Value>::operator = 443, 459
- WCPtrSkipListDict<Key, Value>::operator == 443, 460
- WCPtrSkipListDict<Key, Value>::operator [] 443, 457-458
- WCPtrSkipListDict<Key, Value>::remove 443, 461
- WCPtrSkipListDict<Key, Value>::WCPtrSkipListDict 442-443
- WCPtrSkipListDict<Key, Value>::WCPtrSkipListDict<Key, Value> 444-446
- WCPtrSkipListDict<Key, Value>::~~WCPtrSkipListDict 443
- WCPtrSkipListDict<Key, Value>::~~WCPtrSkipListDict<Key, Value> 447
- WCPtrSkipListSet, member function
  - WCPtrSkipList<Type> 463
  - WCPtrSkipListSet<Type> 463
- WCPtrSkipListSet<Type>::clear 463, 472
- WCPtrSkipListSet<Type>::clearAndDestroy 463, 473
- WCPtrSkipListSet<Type>::contains 463, 474
- WCPtrSkipListSet<Type>::entries 463, 475
- WCPtrSkipListSet<Type>::find 463, 476
- WCPtrSkipListSet<Type>::forall 463, 477
- WCPtrSkipListSet<Type>::insert 463, 478
- WCPtrSkipListSet<Type>::isEmpty 463, 479
- WCPtrSkipListSet<Type>::occurrencesOf 463, 480
- WCPtrSkipListSet<Type>::operator = 463, 481
- WCPtrSkipListSet<Type>::operator == 463, 482
- WCPtrSkipListSet<Type>::remove 463, 483
- WCPtrSkipListSet<Type>::removeAll 463, 484
- WCPtrSkipListSet<Type>::WCPtrSkipList 462-463
- WCPtrSkipListSet<Type>::WCPtrSkipListSet 463
- WCPtrSkipListSet<Type>::~~WCPtrSkipList 463
- WCPtrSkipListSet<Type>::~~WCPtrSkipListSet 463
- WCPtrSList, member function
  - WCPtrDList<Type> 266-267, 269
  - WCPtrSList<Type> 266-267, 269
- WCPtrSList<Type>::append 264, 274
- WCPtrSList<Type>::clear 264, 275
- WCPtrSList<Type>::clearAndDestroy 264, 276
- WCPtrSList<Type>::contains 264, 277
- WCPtrSList<Type>::entries 264, 278
- WCPtrSList<Type>::find 264, 279

- WCPtrSList<Type>::findLast 264, 280
- WCPtrSList<Type>::forAll 264, 281
- WCPtrSList<Type>::get 264, 282
- WCPtrSList<Type>::index 264, 283
- WCPtrSList<Type>::insert 264, 284
- WCPtrSList<Type>::isEmpty 264, 285
- WCPtrSList<Type>::operator = 265, 286
- WCPtrSList<Type>::operator == 265, 287
- WCPtrSList<Type>::WCPtrDList 268, 270-273
- WCPtrSList<Type>::WCPtrSList 266-267, 269
- WCPtrSList<Type>::WCPtrSList<Type> 264
- WCPtrSList<Type>::~~WCPtrSList<Type> 264
- WCPtrSListItemSize
  - macro 431, 531
- WCPtrSListIter, member function
  - WCPtrDListIter<Type> 373-374
  - WCPtrSListIter<Type> 373-374
- WCPtrSListIter<Type>::append 371, 379
- WCPtrSListIter<Type>::container 371, 380
- WCPtrSListIter<Type>::current 371, 381
- WCPtrSListIter<Type>::insert 371-372, 382
- WCPtrSListIter<Type>::operator () 372, 383
- WCPtrSListIter<Type>::operator ++ 372, 384
- WCPtrSListIter<Type>::operator += 372, 385
- WCPtrSListIter<Type>::operator -- 371-372, 386
- WCPtrSListIter<Type>::operator -= 371-372, 387
- WCPtrSListIter<Type>::reset 371, 388-389
- WCPtrSListIter<Type>::WCPtrDListIter 376-377
- WCPtrSListIter<Type>::WCPtrSListIter 373-374
- WCPtrSListIter<Type>::WCPtrSListIter<Type>
  - 371
- WCPtrSListIter<Type>::~~WCPtrDListIter 378
- WCPtrSListIter<Type>::~~WCPtrSListIter 375
- WCPtrSListIter<Type>::~~WCPtrSListIter<Type>
  - 371
- WCPtrSortedVector, member function
  - WCPtrOrderedVector<Type> 541
  - WCPtrSortedVector<Type> 541
- WCPtrSortedVector<Type>::append 542, 549
- WCPtrSortedVector<Type>::clear 541, 550
- WCPtrSortedVector<Type>::clearAndDestroy
  - 541, 551
- WCPtrSortedVector<Type>::contains 541, 552
- WCPtrSortedVector<Type>::entries 541, 553
- WCPtrSortedVector<Type>::find 541, 554
- WCPtrSortedVector<Type>::first 541, 555
- WCPtrSortedVector<Type>::index 541, 556
- WCPtrSortedVector<Type>::insert 541, 557
- WCPtrSortedVector<Type>::insertAt 542, 558
- WCPtrSortedVector<Type>::isEmpty 541, 559
- WCPtrSortedVector<Type>::last 541, 560
- WCPtrSortedVector<Type>::occurrencesOf 541,
  - 561
- WCPtrSortedVector<Type>::operator = 542, 563
- WCPtrSortedVector<Type>::operator == 542,
  - 564
- WCPtrSortedVector<Type>::operator [] 542, 562
- WCPtrSortedVector<Type>::prepend 542, 565
- WCPtrSortedVector<Type>::remove 541, 566
- WCPtrSortedVector<Type>::removeAll 541, 567
- WCPtrSortedVector<Type>::removeAt 541, 568
- WCPtrSortedVector<Type>::removeFirst 541,
  - 569
- WCPtrSortedVector<Type>::removeLast 541,
  - 570
- WCPtrSortedVector<Type>::resize 541, 571
- WCPtrSortedVector<Type>::WCPtrOrderedVecto
  - r 541
- WCPtrSortedVector<Type>::WCPtrSortedVector
  - 541
- WCPtrSortedVector<Type>::WCPtrSortedVector
  - <Type> 543-544, 546-547
- WCPtrSortedVector<Type>::~~WCPtrOrderedVect
  - or 541
- WCPtrSortedVector<Type>::~~WCPtrSortedVecto
  - r 541
- WCPtrSortedVector<Type>::~~WCPtrSortedVecto
  - r<Type> 545, 548
- WCPtrVector<Type>::clear 572, 577
- WCPtrVector<Type>::clearAndDestroy 572, 578
- WCPtrVector<Type>::length 572, 579
- WCPtrVector<Type>::operator = 572, 581
- WCPtrVector<Type>::operator == 572, 582
- WCPtrVector<Type>::operator [] 572, 580
- WCPtrVector<Type>::resize 572, 583
- WCPtrVector<Type>::WCPtrVector<Type>
  - 572-575

- WCPtrVector<Type>::~~WCPtrVector<Type>  
572, 576
- WQueue<Type,FType>::clear 428, 433
- WQueue<Type,FType>::entries 428, 434
- WQueue<Type,FType>::first 428, 435
- WQueue<Type,FType>::get 428, 436
- WQueue<Type,FType>::insert 428, 437
- WQueue<Type,FType>::isEmpty 428, 438
- WQueue<Type,FType>::last 428, 439
- WQueue<Type,FType>::WQueue<Type,FType>  
e> 428, 430-431
- WQueue<Type,FType>::~~WQueue<Type,FType>  
e> 428, 432
- WCSLink 237
- WCSLink::WCSLink 288-289
- WCSLink::~~WCSLink 288, 290
- WStack<Type,FType>::clear 528, 533
- WStack<Type,FType>::entries 528, 534
- WStack<Type,FType>::isEmpty 528, 535
- WStack<Type,FType>::pop 528, 536
- WStack<Type,FType>::push 528, 537
- WStack<Type,FType>::top 528, 538
- WStack<Type,FType>::WStack<Type,FType>  
528, 530-531
- WStack<Type,FType>::~~WStack<Type,FType>  
> 528, 532
- WValConstDListIter, member function
  - WValConstDListIter<Type> 395-396
  - WValConstSListIter<Type> 395-396
- WValConstDListIter<Type>::append 390
- WValConstDListIter<Type>::container 390,  
398
- WValConstDListIter<Type>::current 390, 399
- WValConstDListIter<Type>::insert 390
- WValConstDListIter<Type>::operator () 391,  
400
- WValConstDListIter<Type>::operator ++ 391,  
401
- WValConstDListIter<Type>::operator += 391,  
402
- WValConstDListIter<Type>::operator -- 391,  
403
- WValConstDListIter<Type>::operator -= 391,  
404
- WValConstDListIter<Type>::reset 390-391,  
405-406
- WValConstDListIter<Type>::WValConstDListIter 395-396
- WValConstDListIter<Type>::WValConstSListIter 392-393
- WValConstDListIter<Type>::~~WValConstDListIter 397
- WValConstDListIter<Type>::~~WValConstSListIter 394
- WValConstSListIter, member function
  - WValConstDListIter<Type> 392-393
  - WValConstSListIter<Type> 392-393
- WValConstSListIter<Type>::append 390
- WValConstSListIter<Type>::container 390, 398
- WValConstSListIter<Type>::current 390, 399
- WValConstSListIter<Type>::insert 390
- WValConstSListIter<Type>::operator () 391,  
400
- WValConstSListIter<Type>::operator ++ 391,  
401
- WValConstSListIter<Type>::operator += 391,  
402
- WValConstSListIter<Type>::operator -- 391,  
403
- WValConstSListIter<Type>::operator -= 391,  
404
- WValConstSListIter<Type>::reset 390-391,  
405-406
- WValConstSListIter<Type>::WValConstDListIter 395-396
- WValConstSListIter<Type>::WValConstSListIter 392-393
- WValConstSListIter<Type>::WValConstSListIter<Type> 390
- WValConstSListIter<Type>::~~WValConstDListIter 397
- WValConstSListIter<Type>::~~WValConstSListIter 394
- WValConstSListIter<Type>::~~WValConstSListIter<Type> 390
- WValDList, member function
  - WValDList<Type> 296, 298-301
  - WValSList<Type> 296, 298-301

- WCVaDList<Type>::append 291, 302
- WCVaDList<Type>::clear 291, 303
- WCVaDList<Type>::clearAndDestroy 291, 304
- WCVaDList<Type>::contains 291, 305
- WCVaDList<Type>::entries 292, 306
- WCVaDList<Type>::find 292, 307
- WCVaDList<Type>::findLast 292, 308
- WCVaDList<Type>::forAll 292, 309
- WCVaDList<Type>::get 292, 310
- WCVaDList<Type>::index 292, 311
- WCVaDList<Type>::insert 292, 312
- WCVaDList<Type>::isEmpty 292, 313
- WCVaDList<Type>::operator = 292, 314
- WCVaDList<Type>::operator == 292, 315
- WCVaDList<Type>::WCVaDList 296, 298-301
- WCVaDList<Type>::WCVaSList 294-295, 297
- WCVaDListItemSize
  - macro 271, 299
- WCVaDListIter, member function
  - WCVaDListIter<Type> 413-414
  - WCVaSListIter<Type> 413-414
- WCVaDListIter<Type>::append 407, 416
- WCVaDListIter<Type>::container 407, 417
- WCVaDListIter<Type>::current 407, 418
- WCVaDListIter<Type>::insert 407-408, 419
- WCVaDListIter<Type>::operator () 408, 420
- WCVaDListIter<Type>::operator ++ 408, 421
- WCVaDListIter<Type>::operator += 408, 422
- WCVaDListIter<Type>::operator -- 407-408, 423
- WCVaDListIter<Type>::operator -= 407-408, 424
- WCVaDListIter<Type>::reset 407, 425-426
- WCVaDListIter<Type>::WCVaDListIter 413-414
- WCVaDListIter<Type>::WCVaSListIter 410-411
- WCVaDListIter<Type>::~~WCVaDListIter 415
- WCVaDListIter<Type>::~~WCVaSListIter 412
- WCVaHashDict, member function
  - WCVaHashDict<Key, Value> 138
- WCVaHashDict<Key, Value>::bitHash 138, 143
- WCVaHashDict<Key, Value>::buckets 138, 144
- WCVaHashDict<Key, Value>::clear 138, 145
- WCVaHashDict<Key, Value>::contains 138, 146
- WCVaHashDict<Key, Value>::entries 138, 147
- WCVaHashDict<Key, Value>::find 138, 148
- WCVaHashDict<Key, Value>::findKeyAndValue 138, 149
- WCVaHashDict<Key, Value>::forall 138, 150
- WCVaHashDict<Key, Value>::insert 138, 151
- WCVaHashDict<Key, Value>::isEmpty 138, 152
- WCVaHashDict<Key, Value>::operator = 138, 155
- WCVaHashDict<Key, Value>::operator == 138, 156
- WCVaHashDict<Key, Value>::operator [] 138, 153-154
- WCVaHashDict<Key, Value>::remove 138, 157
- WCVaHashDict<Key, Value>::resize 138, 158
- WCVaHashDict<Key, Value>::WCVaHashDict 138
- WCVaHashDict<Key, Value>::WCVaHashDict<Key, Value> 139-141
- WCVaHashDict<Key, Value>::~~WCVaHashDict 138
- WCVaHashDict<Key, Value>::~~WCVaHashDict<Key, Value> 142
- WCVaHashDictIter, member function
  - WCVaHashDictIter<Key, Value> 198-199
- WCVaHashDictIter<Key, Value>::container 197, 201
- WCVaHashDictIter<Key, Value>::key 197, 202
- WCVaHashDictIter<Key, Value>::operator () 197, 203
- WCVaHashDictIter<Key, Value>::operator ++ 197, 204
- WCVaHashDictIter<Key, Value>::reset 197, 205-206
- WCVaHashDictIter<Key, Value>::value 197, 207
- WCVaHashDictIter<Key, Value>::WCVaHashDictIter 198-199
- WCVaHashDictIter<Key, Value>::WCVaHashDictIter<Key, Value> 197
- WCVaHashDictIter<Key, Value>::~~WCVaHashDictIter 200

- WCValHashDictIter<Key, Value>::~WCValHashDictIter<Key, Value> 197
- WCValHashSet, member function
  - WCValHashSet<Type> 160
  - WCValHashTable<Type> 160
- WCValHashSet<Type>::bitHash 160, 169
- WCValHashSet<Type>::buckets 160, 170
- WCValHashSet<Type>::clear 160, 171
- WCValHashSet<Type>::contains 160, 172
- WCValHashSet<Type>::entries 160, 173
- WCValHashSet<Type>::find 160, 174
- WCValHashSet<Type>::forall 160, 175
- WCValHashSet<Type>::insert 160, 176
- WCValHashSet<Type>::isEmpty 160, 177
- WCValHashSet<Type>::occurrencesOf 160, 178
- WCValHashSet<Type>::operator = 160, 179
- WCValHashSet<Type>::operator == 160, 180
- WCValHashSet<Type>::remove 160, 181
- WCValHashSet<Type>::removeAll 160, 182
- WCValHashSet<Type>::resize 160, 183
- WCValHashSet<Type>::WCValHashSet 160
- WCValHashSet<Type>::WCValHashTable 160
- WCValHashSet<Type>::~WCValHashSet 160
- WCValHashSet<Type>::~WCValHashTable 160
- WCValHashSetIter, member function
  - WCValHashSetIter<Type> 222-223
  - WCValHashTableIter<Type> 222-223
- WCValHashSetIter<Type>::container 221, 228
- WCValHashSetIter<Type>::current 221, 229
- WCValHashSetIter<Type>::operator () 221, 230
- WCValHashSetIter<Type>::operator ++ 221, 231
- WCValHashSetIter<Type>::reset 221, 232-233
- WCValHashSetIter<Type>::WCValHashSetIter 222-223
- WCValHashSetIter<Type>::WCValHashSetIter<Type> 221
- WCValHashSetIter<Type>::WCValHashTableIter 225-226
- WCValHashSetIter<Type>::~WCValHashSetIter 224
- WCValHashSetIter<Type>::~WCValHashSetIter<Type> 221
- WCValHashSetIter<Type>::~WCValHashTableIter 227
- WCValHashTable, member function
  - WCValHashSet<Type> 160
  - WCValHashTable<Type> 160
- WCValHashTable<Type>::bitHash 160, 169
- WCValHashTable<Type>::buckets 160, 170
- WCValHashTable<Type>::clear 160, 171
- WCValHashTable<Type>::contains 160, 172
- WCValHashTable<Type>::entries 160, 173
- WCValHashTable<Type>::find 160, 174
- WCValHashTable<Type>::forall 160, 175
- WCValHashTable<Type>::insert 160, 176
- WCValHashTable<Type>::isEmpty 160, 177
- WCValHashTable<Type>::occurrencesOf 160, 178
- WCValHashTable<Type>::operator = 160, 179
- WCValHashTable<Type>::operator == 160, 180
- WCValHashTable<Type>::remove 160, 181
- WCValHashTable<Type>::removeAll 160, 182
- WCValHashTable<Type>::resize 160, 183
- WCValHashTable<Type>::WCValHashSet 160
- WCValHashTable<Type>::WCValHashTable 160
- WCValHashTable<Type>::WCValHashTable<Type> 161-163, 165-167
- WCValHashTable<Type>::~WCValHashSet 160
- WCValHashTable<Type>::~WCValHashTable 160
- WCValHashTable<Type>::~WCValHashTable<Type> 164, 168
- WCValHashTableIter, member function
  - WCValHashSetIter<Type> 225-226
  - WCValHashTableIter<Type> 225-226
- WCValHashTableIter<Type>::container 221, 228
- WCValHashTableIter<Type>::current 221, 229
- WCValHashTableIter<Type>::operator () 221, 230
- WCValHashTableIter<Type>::operator ++ 221, 231
- WCValHashTableIter<Type>::reset 221, 232-233
- WCValHashTableIter<Type>::WCValHashSetIter 222-223
- WCValHashTableIter<Type>::WCValHashTableIter 225-226



- 
- WCValHashTableIter<Type>::~~WCValHashSetIter 224
  - WCValHashTableIter<Type>::~~WCValHashTableIter 227
  - WCValOrderedVector, member function
    - WCValOrderedVector<Type> 585
    - WCValSortedVector<Type> 585
  - WCValOrderedVector<Type>::append 586, 593
  - WCValOrderedVector<Type>::clear 585, 594
  - WCValOrderedVector<Type>::contains 585, 595
  - WCValOrderedVector<Type>::entries 585, 596
  - WCValOrderedVector<Type>::find 585, 597
  - WCValOrderedVector<Type>::first 586, 598
  - WCValOrderedVector<Type>::index 586, 599
  - WCValOrderedVector<Type>::insert 586, 600
  - WCValOrderedVector<Type>::insertAt 586, 601
  - WCValOrderedVector<Type>::isEmpty 586, 602
  - WCValOrderedVector<Type>::last 586, 603
  - WCValOrderedVector<Type>::occurrencesOf 586, 604
  - WCValOrderedVector<Type>::operator = 586, 606
  - WCValOrderedVector<Type>::operator == 586, 607
  - WCValOrderedVector<Type>::operator [] 586, 605
  - WCValOrderedVector<Type>::prepend 586, 608
  - WCValOrderedVector<Type>::remove 586, 609
  - WCValOrderedVector<Type>::removeAll 586, 610
  - WCValOrderedVector<Type>::removeAt 586, 611
  - WCValOrderedVector<Type>::removeFirst 586, 612
  - WCValOrderedVector<Type>::removeLast 586, 613
  - WCValOrderedVector<Type>::resize 586, 614
  - WCValOrderedVector<Type>::WCValOrderedVector 585
  - WCValOrderedVector<Type>::WCValSortedVector 585
  - WCValOrderedVector<Type>::~~WCValOrderedVector 585
  - WCValOrderedVector<Type>::~~WCValSortedVector 585
  - WCValSkipList, member function
    - WCValSkipList<Type> 505
    - WCValSkipListSet<Type> 505
  - WCValSkipList<Type>::clear 505, 514
  - WCValSkipList<Type>::contains 505, 515
  - WCValSkipList<Type>::entries 505, 516
  - WCValSkipList<Type>::find 505, 517
  - WCValSkipList<Type>::forall 505, 518
  - WCValSkipList<Type>::insert 505, 519
  - WCValSkipList<Type>::isEmpty 505, 520
  - WCValSkipList<Type>::occurrencesOf 505, 521
  - WCValSkipList<Type>::operator = 505, 522
  - WCValSkipList<Type>::operator == 505, 523
  - WCValSkipList<Type>::remove 505, 524
  - WCValSkipList<Type>::removeAll 505, 525
  - WCValSkipList<Type>::WCValSkipList 505
  - WCValSkipList<Type>::WCValSkipList<Type> 506-508, 510-512
  - WCValSkipList<Type>::WCValSkipListSet 505
  - WCValSkipList<Type>::~~WCValSkipList 505
  - WCValSkipList<Type>::~~WCValSkipList<Type> 509, 513
  - WCValSkipList<Type>::~~WCValSkipListSet 505
  - WCValSkipListDict, member function
    - WCValSkipListDict<Key, Value> 486
  - WCValSkipListDict<Key, Value>::clear 486, 491
  - WCValSkipListDict<Key, Value>::contains 486, 492
  - WCValSkipListDict<Key, Value>::entries 486, 493
  - WCValSkipListDict<Key, Value>::find 486, 494
  - WCValSkipListDict<Key, Value>::findKeyAndValue 486, 495
  - WCValSkipListDict<Key, Value>::forall 486, 496
  - WCValSkipListDict<Key, Value>::insert 486, 497
  - WCValSkipListDict<Key, Value>::isEmpty 486, 498
  - WCValSkipListDict<Key, Value>::operator = 486, 501

- WCValSkipListDict<Key, Value>::operator == 486, 502
- WCValSkipListDict<Key, Value>::operator [] 486, 499-500
- WCValSkipListDict<Key, Value>::remove 486, 503
- WCValSkipListDict<Key, Value>::WCValSkipListDict 486
- WCValSkipListDict<Key, Value>::WCValSkipListDict<Key, Value> 487-489
- WCValSkipListDict<Key, Value>::~WCValSkipListDict 486
- WCValSkipListDict<Key, Value>::~WCValSkipListDict<Key, Value> 490
- WCValSkipListSet, member function
  - WCValSkipList<Type> 505
  - WCValSkipListSet<Type> 505
- WCValSkipListSet<Type>::clear 505, 514
- WCValSkipListSet<Type>::contains 505, 515
- WCValSkipListSet<Type>::entries 505, 516
- WCValSkipListSet<Type>::find 505, 517
- WCValSkipListSet<Type>::forall 505, 518
- WCValSkipListSet<Type>::insert 505, 519
- WCValSkipListSet<Type>::isEmpty 505, 520
- WCValSkipListSet<Type>::occurrencesOf 505, 521
- WCValSkipListSet<Type>::operator = 505, 522
- WCValSkipListSet<Type>::operator == 505, 523
- WCValSkipListSet<Type>::remove 505, 524
- WCValSkipListSet<Type>::removeAll 505, 525
- WCValSkipListSet<Type>::WCValSkipList 505
- WCValSkipListSet<Type>::WCValSkipListSet 505
- WCValSkipListSet<Type>::~WCValSkipList 505
- WCValSkipListSet<Type>::~WCValSkipListSet 505
- WCValSList, member function
  - WCValDList<Type> 294-295, 297
  - WCValSList<Type> 294-295, 297
- WCValSList<Type>::append 291, 302
- WCValSList<Type>::clear 291, 303
- WCValSList<Type>::clearAndDestroy 291, 304
- WCValSList<Type>::contains 291, 305
- WCValSList<Type>::entries 292, 306
- WCValSList<Type>::find 292, 307
- WCValSList<Type>::findLast 292, 308
- WCValSList<Type>::forall 292, 309
- WCValSList<Type>::get 292, 310
- WCValSList<Type>::index 292, 311
- WCValSList<Type>::insert 292, 312
- WCValSList<Type>::isEmpty 292, 313
- WCValSList<Type>::operator = 292, 314
- WCValSList<Type>::operator == 292, 315
- WCValSList<Type>::WCValDList 296, 298-301
- WCValSList<Type>::WCValSList 294-295, 297
- WCValSList<Type>::WCValSList<Type> 291
- WCValSList<Type>::~WCValSList<Type> 291
- WCValSListItemSize
  - macro 267, 295, 431, 531
- WCValSListIter, member function
  - WCValDListIter<Type> 410-411
  - WCValSListIter<Type> 410-411
- WCValSListIter<Type>::append 407, 416
- WCValSListIter<Type>::container 407, 417
- WCValSListIter<Type>::current 407, 418
- WCValSListIter<Type>::insert 407-408, 419
- WCValSListIter<Type>::operator () 408, 420
- WCValSListIter<Type>::operator ++ 408, 421
- WCValSListIter<Type>::operator += 408, 422
- WCValSListIter<Type>::operator -- 407-408, 423
- WCValSListIter<Type>::operator -= 407-408, 424
- WCValSListIter<Type>::reset 407, 425-426
- WCValSListIter<Type>::WCValDListIter 413-414
- WCValSListIter<Type>::WCValSListIter 410-411
- WCValSListIter<Type>::WCValSListIter<Type> 407
- WCValSListIter<Type>::~WCValDListIter 415
- WCValSListIter<Type>::~WCValSListIter 412
- WCValSListIter<Type>::~WCValSListIter<Type> > 407
- WCValSortedVector, member function
  - WCValOrderedVector<Type> 585
  - WCValSortedVector<Type> 585

WCValsortedVector<Type>::append 586, 593  
 WCValsortedVector<Type>::clear 585, 594  
 WCValsortedVector<Type>::contains 585, 595  
 WCValsortedVector<Type>::entries 585, 596  
 WCValsortedVector<Type>::find 585, 597  
 WCValsortedVector<Type>::first 586, 598  
 WCValsortedVector<Type>::index 586, 599  
 WCValsortedVector<Type>::insert 586, 600  
 WCValsortedVector<Type>::insertAt 586, 601  
 WCValsortedVector<Type>::isEmpty 586, 602  
 WCValsortedVector<Type>::last 586, 603  
 WCValsortedVector<Type>::occurrencesOf 586,  
 604  
 WCValsortedVector<Type>::operator = 586, 606  
 WCValsortedVector<Type>::operator == 586,  
 607  
 WCValsortedVector<Type>::operator [] 586,  
 605  
 WCValsortedVector<Type>::prepend 586, 608  
 WCValsortedVector<Type>::remove 586, 609  
 WCValsortedVector<Type>::removeAll 586,  
 610  
 WCValsortedVector<Type>::removeAt 586, 611  
 WCValsortedVector<Type>::removeFirst 586,  
 612  
 WCValsortedVector<Type>::removeLast 586,  
 613  
 WCValsortedVector<Type>::resize 586, 614  
 WCValsortedVector<Type>::WCValOrderedVect  
 or 585  
 WCValsortedVector<Type>::WCValSortedVecto  
 r 585  
 WCValsortedVector<Type>::WCValSortedVecto  
 r<Type> 587-588, 590-591  
 WCValsortedVector<Type>::~WCValOrderedVe  
 ctor 585  
 WCValsortedVector<Type>::~WCValSortedVect  
 or 585  
 WCValsortedVector<Type>::~WCValSortedVect  
 or<Type> 589, 592  
 WCValVector<Type>::clear 616, 621  
 WCValVector<Type>::length 616, 622  
 WCValVector<Type>::operator = 616, 624  
 WCValVector<Type>::operator == 616, 625

WCValVector<Type>::operator [] 616, 623  
 WCValVector<Type>::resize 616, 626  
 WCValVector<Type>::WCValVector<Type>  
 615, 617-619  
 WCValVector<Type>::~WCValVector<Type>  
 615, 620  
 width, member function  
 ios 674, 708  
 write, member function  
 ostream 775, 797  
 ws, manipulator 752, 766

X

xalloc, member function  
 ios 674, 709

Z

zero\_buckets  
 exception 75, 110, 136, 158, 183  
 zero\_buckets, member enumeration  
 WCEXCEPT 74

~

~WCISVConstDListIter, member function  
 WCISVConstDListIter<Type> 325  
 WCISVConstSListIter<Type> 325  
 ~WCISVConstSListIter, member function  
 WCISVConstDListIter<Type> 322  
 WCISVConstSListIter<Type> 322  
 ~WCISVDListIter, member function

- WCIsvDListIter<Type> 342
- WCIsvSListIter<Type> 342
- ~WCIsvSListIter, member function
  - WCIsvDListIter<Type> 339
  - WCIsvSListIter<Type> 339
- ~WCPtrConstDListIter, member function
  - WCPtrConstDListIter<Type> 361
  - WCPtrConstSListIter<Type> 361
- ~WCPtrConstSListIter, member function
  - WCPtrConstDListIter<Type> 358
  - WCPtrConstSListIter<Type> 358
- ~WCPtrDListIter, member function
  - WCPtrDListIter<Type> 378
  - WCPtrSListIter<Type> 378
- ~WCPtrHashDict, member function
  - WCPtrHashDict<Key, Value> 89
- ~WCPtrHashDictIter, member function
  - WCPtrHashDictIter<Key, Value> 189
- ~WCPtrHashSet, member function
  - WCPtrHashSet<Type> 112
  - WCPtrHashTable<Type> 112
- ~WCPtrHashSetIter, member function
  - WCPtrHashSetIter<Type> 211
  - WCPtrHashTableIter<Type> 211
- ~WCPtrHashTable, member function
  - WCPtrHashSet<Type> 112
  - WCPtrHashTable<Type> 112
- ~WCPtrHashTableIter, member function
  - WCPtrHashSetIter<Type> 214
  - WCPtrHashTableIter<Type> 214
- ~WCPtrOrderedVector, member function
  - WCPtrOrderedVector<Type> 541
  - WCPtrSortedVector<Type> 541
- ~WCPtrSkipList, member function
  - WCPtrSkipList<Type> 463
  - WCPtrSkipListSet<Type> 463
- ~WCPtrSkipListDict, member function
  - WCPtrSkipListDict<Key, Value> 443
- ~WCPtrSkipListSet, member function
  - WCPtrSkipList<Type> 463
  - WCPtrSkipListSet<Type> 463
- ~WCPtrSListIter, member function
  - WCPtrDListIter<Type> 375
  - WCPtrSListIter<Type> 375
- ~WCPtrSortedVector, member function
  - WCPtrOrderedVector<Type> 541
  - WCPtrSortedVector<Type> 541
- ~WCValConstDListIter, member function
  - WCValConstDListIter<Type> 397
  - WCValConstSListIter<Type> 397
- ~WCValConstSListIter, member function
  - WCValConstDListIter<Type> 394
  - WCValConstSListIter<Type> 394
- ~WCValDListIter, member function
  - WCValDListIter<Type> 415
  - WCValSListIter<Type> 415
- ~WCValHashDict, member function
  - WCValHashDict<Key, Value> 138
- ~WCValHashDictIter, member function
  - WCValHashDictIter<Key, Value> 200
- ~WCValHashSet, member function
  - WCValHashSet<Type> 160
  - WCValHashTable<Type> 160
- ~WCValHashSetIter, member function
  - WCValHashSetIter<Type> 224
  - WCValHashTableIter<Type> 224
- ~WCValHashTable, member function
  - WCValHashSet<Type> 160
  - WCValHashTable<Type> 160
- ~WCValHashTableIter, member function
  - WCValHashSetIter<Type> 227
  - WCValHashTableIter<Type> 227
- ~WCValOrderedVector, member function
  - WCValOrderedVector<Type> 585
  - WCValSortedVector<Type> 585
- ~WCValSkipList, member function
  - WCValSkipList<Type> 505
  - WCValSkipListSet<Type> 505
- ~WCValSkipListDict, member function
  - WCValSkipListDict<Key, Value> 486
- ~WCValSkipListSet, member function
  - WCValSkipList<Type> 505
  - WCValSkipListSet<Type> 505
- ~WCValSListIter, member function
  - WCValDListIter<Type> 412
  - WCValSListIter<Type> 412
- ~WCValSortedVector, member function
  - WCValOrderedVector<Type> 585

WCValSortedVector<Type> 585