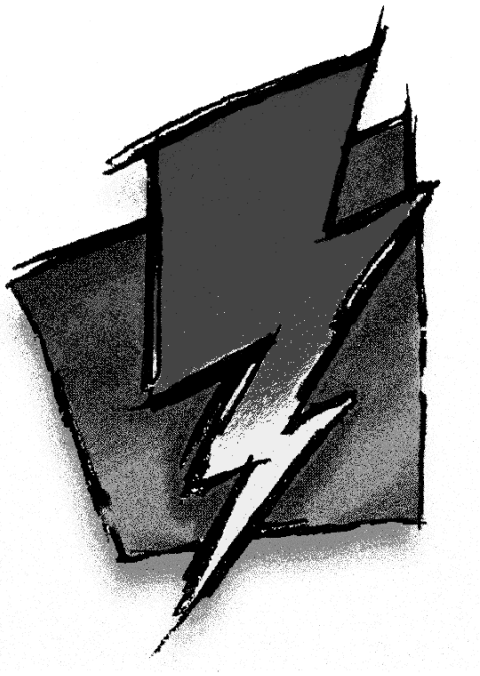
Watcom C/C++

User's Guide



Edition 11.0c

Notice of Copyright

Printed in U.S.A.

Preface

Watcom C is an implementation of ANSI/ISO 9899:1990 Programming Language C. The standard was developed by the ANSI X3J11 Technical Committee on the C Programming Language. In addition to the full C language standard, the compiler supports numerous extensions for the Intel 80x86-based personal computer environment.

Watcom C++ is an implementation of the Draft Proposed International Standard for Information Systems Programming Language C++ (ANSI X3J16, ISO WG21). In addition to the full C++ language standard, the compiler supports numerous extensions for the Intel 80x86-based personal computer environment.

Watcom is well known for its language processors having developed, over the last decade, compilers and interpreters for the APL, BASIC, COBOL, FORTRAN and Pascal programming languages. From the start, Watcom has been committed to developing portable software products. These products have been implemented on a variety of processor architectures including the IBM 370, the Intel 8086 family, the Motorola 6809 and 68000, the MOS 6502, and the Digital PDP11 and VAX. In most cases, the tools necessary for porting to these environments had to be created first. Invariably, a code generator had to be written. Assemblers, linkers and debuggers had to be created when none were available or when existing ones were inadequate.

Over the years, much research has gone into developing the "ultimate" code generator for the Intel 8086 family. We have continually looked for new ways to improve the quality of the emitted code, never being quite satisfied with the results. Several major revisions, including some entirely new approaches to code generation, have ensued over the years. Our latest version employs state of the art techniques to produce very high quality code for the 8086 family. We introduced the C compiler in 1987, satisfied that we had a C software development system that would be of major benefit to those developing applications in C for the IBM PC and compatibles.

The *Watcom C/C++ User's Guide* describes how to use Watcom C/C++ on Intel 80x86-based personal computers with DOS, Windows, Windows NT, or OS/2.

Acknowledgements

This book was produced with the 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 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.

The Plum Hall Validation Suite for C/C++ has been invaluable in verifying the conformance of the Watcom C/C++ compilers to the ANSI C Language Standard and the Draft Proposed C++ Language Standard.

Many users have provided valuable feedback on earlier versions of the Watcom C/C++ compilers and related tools. Their comments were greatly appreciated. If you find problems in the documentation or have some good suggestions, we would like to hear from you.

September, 2000.

Trademarks Used in this Manual

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Watcom C/C++ User's Guide

1 About This Manual

This manual contains the following chapters:

Chapter 1 — "About This Manual".

This chapter provides an overview of the contents of this guide.

Chapter 2 — "Watcom C/C++ Compiler Options" on page 7.

This chapter provides a summary and reference section for all the C and C++ compiler options.

Chapter 3 — "The Watcom C/C++ Compilers" on page 75.

This chapter describes how to compile an application from the command line. This chapter also describes compiler environment variables, benchmarking hints, compiler diagnostics, #include file processing, the preprocessor, predefined macros, extended keywords, and the code generator.

Chapter 4 — "Precompiled Headers" on page 117.

This chapter describes the use of precompiled headers to speed up compilation.

Chapter 5 — "The Watcom C/C++ Libraries" on page 121.

This chapter describes the Watcom C/C++ library directory structure, C libraries, class libraries, math libraries, 80x87 math libraries, alternate math libraries, the "NO87" environment variable, and the run-time initialization routines.

Chapter 6 — "16-bit Memory Models" on page 135.

This chapter describes the Watcom C/C++ memory models (including code and data models), the tiny memory model, the mixed memory model, linking applications for the various memory models, creating a tiny memory model application, and memory layout in an executable.

Chapter 7 — "16-bit Assembly Language Considerations" on page 141.

This chapter describes issues relating to 16-bit interfacing such as parameter passing conventions.

Chapter 8 — "16-bit Pragmas" on page 161.

This chapter describes the use of pragmas with the 16-bit compilers.

Chapter 9 — "32-bit Memory Models" on page 217.

This chapter describes the Watcom C/C++ memory models (including code and data models), the flat memory model, the mixed memory model, linking applications for the various memory models, and memory layout in an executable.

Chapter 10 — "32-bit Assembly Language Considerations" on page 223.

This chapter describes issues relating to 32-bit interfacing such as parameter passing conventions.

Chapter 11 — "32-bit Pragmas" on page 247.

This chapter describes the use of pragmas with the 32-bit compilers.

Chapter 12 — "In-line Assembly Language" on page 305.

This chapter describes in-line assembly language programming using the auxiliary pragma.

Chapter 13 — "Creating ROM-based Applications" on page 343.

This chapter discusses some embedded systems issues as they pertain to the C library.

Appendix A. — "Use of Environment Variables" on page 351.

This appendix describes all the environment variables used by the compilers and related tools.

Appendix B. — "Watcom C Diagnostic Messages" on page 363.

This appendix lists all of the Watcom C diagnostic messages with an explanation for each.

4 About This Manual

Appendix C. — "Watcom C++ Diagnostic Messages" on page 397.

This appendix lists all of the Watcom C++ diagnostic messages with an explanation for each.

Appendix D. — "Watcom C/C++ Run-Time Messages" on page 639.

This appendix lists all of the C/C++ run-time diagnostic messages with an explanation for each.

2 Watcom C/C++ Compiler Options

Source files can be compiled using either the IDE, command-line compilers or IBM WorkFrame/2 (OS/2 only). This chapter describes all the compiler options that are available.

For information about compiling applications from the IDE, see the ***Watcom Graphical Tools User's Guide.***

For information about compiling applications from the command line, see the chapter entitled "The Watcom C/C++ Compilers" on page 75.

For information about creating applications using IBM WorkFrame/2, refer to IBM's ***OS/2 Programming Guide*** for more information.

The Watcom C/C++ compiler command names (*compiler_name*) are:

WCC	the Watcom C compiler for 16-bit Intel platforms.
WPP	the Watcom C++ compiler for 16-bit Intel platforms.
WCC386	the Watcom C compiler for 32-bit Intel platforms.
WPP386	the Watcom C++ compiler for 32-bit Intel platforms.

2.1 Compiler Options - Summarized Alphabetically

In this section, we present a terse summary of compiler options. This summary is displayed on the screen by simply entering the compiler command name with no arguments.

Option:	***Description:***
0	(16-bit only) 8088 and 8086 instructions (default for 16-bit) (see "0" on page 60)
1	(16-bit only) 188 and 186 instructions (see "1" on page 60)
2	(16-bit only) 286 instructions (see "2" on page 60)
3	(16-bit only) 386 instructions (see "3" on page 61)
4	(16-bit only) 486 instructions (see "4" on page 61)
5	(16-bit only) Pentium instructions (see "5" on page 61)

6	(16-bit only) Pentium Pro instructions (see "6" on page 61)
3r	(32-bit only) generate 386 instructions based on 386 instruction timings and use register-based argument passing conventions (see "3{r|s}" on page 61)
3s	(32-bit only) generate 386 instructions based on 386 instruction timings and use stack-based argument passing conventions (see "3{r|s}" on page 61)
4r	(32-bit only) generate 386 instructions based on 486 instruction timings and use register-based argument passing conventions (see "4{r|s}" on page 63)
4s	(32-bit only) generate 386 instructions based on 486 instruction timings and use stack-based argument passing conventions (see "4{r|s}" on page 63)
5r	(32-bit only) generate 386 instructions based on Intel Pentium instruction timings and use register-based argument passing conventions (default for 32-bit) (see "5{r|s}" on page 63)
5s	(32-bit only) generate 386 instructions based on Intel Pentium instruction timings and use stack-based argument passing conventions (see "5{r|s}" on page 63)
6r	(32-bit only) generate 386 instructions based on Intel Pentium Pro instruction timings and use register-based argument passing conventions (see "6{r|s}" on page 63)
6s	(32-bit only) generate 386 instructions based on Intel Pentium Pro instruction timings and use stack-based argument passing conventions (see "6{r|s}" on page 63)
bc	(C++ only) build target is a console application (see "bc" on page 20)
bd	build target is a Dynamic Link Library (DLL) (see "bd" on page 20)
bg	(C++ only) build target is a GUI application (see "bg" on page 20)
bm	build target is a multi-thread environment (see "bm" on page 21)
br	build target uses DLL version of C/C++ run-time libraries (see "br" on page 21)
bt[=<os>]	build target for operating system <os> (see "bt[=<os>]" on page 21)
bw	build target uses default windowing support (see "bw" on page 22)
d0	(C++ only) no debugging information (see "d0" on page 28)
d1	line number debugging information (see "d1" on page 29)
d1+	(C only) line number debugging information plus typing information for global symbols and local structs and arrays (see "d1+" on page 29)
d2	full symbolic debugging information (see "d2" on page 29)
d2i	(C++ only) d2 and debug inlines; emit inlines as external out-of-line functions (see "d2i" on page 29)
d2s	(C++ only) d2 and debug inlines; emit inlines as static out-of-line functions (see "d2s" on page 29)
d2t	(C++ only) full symbolic debugging information, without type names (see "d2t" on page 30)
d3	full symbolic debugging with unreferenced type names (see "d3" on page 30) ,*

8 Compiler Options - Summarized Alphabetically

d3i	(C++ only) d3 plus debug inlines; emit inlines as external out-of-line functions (see "d3i" on page 30)
d3s	(C++ only) d3 plus debug inlines; emit inlines as static out-of-line functions (see "d3s" on page 30)
d<name>[=text]	preprocessor #define name [text] (see "d<name>[=text]" on page 32)
d+	allow extended -d macro definitions (see "d+" on page 33)
db	generate browsing information (see "db" on page 40)
e<number>	set error limit number (default is 20) (see "e<number>" on page 35)
ee	call epilogue hook routine (see "ee" on page 30)
ef	use full path names in error messages (see "ef" on page 35)
ei	force enum base type to use at least an int (see "ei" on page 44)
em	force enum base type to use minimum (see "em" on page 44)
en	emit routine name before prologue (see "en" on page 30)
ep[<number>]	call prologue hook routine with number of stack bytes available (see "ep[<number>]" on page 31)
eq	do not display error messages (they are still written to a file) (see "eq" on page 35)
er	(C++ only) do not recover from undefined symbol errors (see "er" on page 35)
et	Pentium profiling (see "et" on page 31)
ew	(C++ only) generate less verbose messages (see "ew" on page 35)
ez	(32-bit only) generate Phar Lap Easy OMF-386 object file (see "ez" on page 40)
fc=<file_name>	(C++ only) specify file of command lines to be batch processed (see "fc=<file_name>" on page 40)
fh[q][=<file_name>]	use precompiled headers (see "fh[q][=<file_name>]" on page 41)
fhd	store debug info for pre-compiled header once (DWARF only) (see "fhd" on page 41)
fhr	(C++ only) force compiler to read pre-compiled header (see "fhr" on page 41)
fhw	(C++ only) force compiler to write pre-compiled header (see "fhw" on page 41)
fhwe	(C++ only) don’t include pre-compiled header warnings when "we" is used (see "fhwe" on page 41)
fi=<file_name>	force file_name to be included (see "fi=<file_name>" on page 41)
fo=<file_name>	set object or preprocessor output file specification (see "fo[=<file_name>] (preprocessor)" on page 33) (see "fo[=<file_name>]" on page 41)
fpc	generate calls to floating-point library (see "fpc" on page 52)
fpi	(16-bit only) generate in-line 80x87 instructions with emulation (default) (32-bit only) generate in-line 387 instructions with emulation (default) (see "fpi" on page 52)

fpi87	(16-bit only) generate in-line 80x87 instructions (32-bit only) generate in-line 387 instructions (see "fpi87" on page 53)
fp2	generate in-line 80x87 instructions (see "fp2" on page 54)
fp3	generate in-line 387 instructions (see "fp3" on page 54)
fp5	generate in-line 80x87 instructions optimized for Pentium processor (see "fp5" on page 54)
fp6	generate in-line 80x87 instructions optimized for Pentium Pro processor (see "fp6" on page 54)
fpd	enable generation of Pentium FDIV bug check code (see "fpd" on page 54)
fpr	generate 8087 code compatible with older versions of compiler (see "fpr" on page 74)
fr=<file_name>	set error file specification (see "fr[=<file_name>]" on page 42)
ft	(C++ only) try truncated (8.3) header file specification (see "ft" on page 42)
fx	(C++ only) do not try truncated (8.3) header file specification (see "fx" on page 42)
g=<codegroup>	set code group name (see "g=<codegroup>" on page 55)
h{w,d,c}	set debug output format (Watcom, Dwarf, Codeview) (see "h{w,d,c}" on page 32)
i=<directory>	add directory to list of include directories (see "i=<directory>" on page 43)
j	change char default from unsigned to signed (see "j" on page 44)
k	(C++ only) continue processing files (ignore errors) (see "k" on page 43)
m{f,s,m,c,l,h}	memory model — mf=flat (see "mf" on page 63), ms=small (see "ms" on page 63), mm=medium (see "mm" on page 64), mc=compact (see "mc" on page 64), ml=large (see "ml" on page 64), mh=huge (see "mh" on page 64) (default is "ms" for 16-bit and Netware, "mf" for 32-bit)
nc=<name>	set name of the code class (see "nc=<name>" on page 56)
nd=<name>	set name of the "data" segment (see "nd=<name>" on page 56)
nm=<name>	set module name different from filename (see "nm=<name>" on page 57)
nt=<name>	set name of the "text" segment (see "nt=<name>" on page 58)
o{a,b,c,d,e,f,f+,h,i,i+,k,l,l+,m,n,o,p,r,s,t,u,x,z}	control optimization (see "oa" on page 66) (see "of" on page 22)
p{e,l,c,w=<num>}	preprocess file only, sending output to standard output; "c" include comments; "e" encrypt identifiers (C++ only); "l" include #line directives; w=<num> wrap output lines at <num> columns (zero means no wrap) (see "p{e,l,c,w=<num>}" on page 34)
r	save/restore segment registers (see "r" on page 74)
ri	return chars and shorts as ints (see "ri" on page 45)
s	remove stack overflow checks (see "s" on page 32)
sg	generate calls to grow the stack (see "sg" on page 24)
st	touch stack through SS first (see "st" on page 25)
t=<num>	(C++ only) set tab stop multiplier (see "t=<num>" on page 35)

10 Compiler Options - Summarized Alphabetically

u<name>	preprocessor #undef name (see "u<name>" on page 34)
v	output function declarations to .def file (with typedef names) (see "v" on page 43)
vc...	(C++ only) VC++ compatibility options (see "vc..." on page 73)
w<number>	set warning level number (default is w1) (see "w<number>" on page 36)
wcd=<num>	warning control: disable warning message <num> (see "wcd=<number>" on page 36)
wce=<num>	warning control: enable warning message <num> (see "wce=<number>" on page 36)
we	treat all warnings as errors (see "we" on page 36)
wo	(C only) (16-bit only) warn about problems with overlaid code (see "wo" on page 36)
wx	set warning level to maximum setting (see "wx" on page 36)
xd	(C++ only) disable exception handling (default) (see "xd" on page 71)
xdt	(C++ only) disable exception handling (same as "xd") (see "xdt" on page 71)
xds	(C++ only) disable exception handling (table-driven destructors) (see "xds" on page 71)
xr	(C++ only) enable RTTI (see "xr" on page 45)
xs	(C++ only) enable exception handling (see "xs" on page 72)
xst	(C++ only) enable exception handling (direct calls for destruction) (see "xst" on page 72)
xss	(C++ only) enable exception handling (table-driven destructors) (see "xss" on page 72)
z{a,e}	disable/enable language extensions (default is ze) (see "za" on page 36) (see "ze" on page 37)
zc	place literal strings in code segment (see "zc" on page 45)
zd{f,p}	allow DS register to "float" or "peg" it to DGROUP (default is zdp) (see "zd{f,p}" on page 65)
zdl	(32-bit only) load DS register directly from DGROUP (see "zdl" on page 65)
zf{f,p}	allow FS register to be used (default for all but flat memory model) or not be used (default for flat memory model) (see "zf{f,p}" on page 65)
zg	output function declarations to .def (without typedef names) (see "zg" on page 43)
zg{f,p}	allow GS register to be used or not used (see "zg{f,p}" on page 65)
zk0	double-byte char support for Kanji (see "zk{0,1,2,l}" on page 72)
zk0u	translate Kanji double-byte characters to UNICODE (see "zk0u" on page 73)
zk1	double-byte char support for Chinese/Taiwanese (see "zk{0,1,2,l}" on page 72)
zk2	double-byte char support for Korean (see "zk{0,1,2,l}" on page 72)

zkl	double-byte char support if current code page has lead bytes (see "zk{0,1,2,l}" on page 72)
zku=<codepage>	load UNICODE translate table for specified code page (see "zku=<codepage>" on page 73)
zl	suppress generation of library file names and references in object file (see "zl" on page 43)
zld	suppress generation of file dependency information in object file (see "zld" on page 44)
zm	place each function in separate segment (near functions not allowed) (see "zm" on page 58)
zmf	place each function in separate segment (near functions allowed) (see "zmf" on page 59)
zp[{1,2,4,8,16}]	set minimal structure packing (member alignment) (default is zp1) (see "zp[{1,2,4,8,16}]" on page 45)
zpw	output warning when padding is added in a struct/class (see "zpw" on page 48)
zq	operate quietly (see "zq" on page 39)
zs	syntax check only (see "zs" on page 40)
zt<number>	set data threshold (default is zt32767) (see "zt<number>" on page 48)
zu	do not assume that SS contains segment of DGROUP (see "zu" on page 65)
zv	(C++ only) enable virtual function removal optimization (see "zv" on page 49)
zw	Microsoft Windows prologue/epilogue code sequences (see "zw" on page 26)
zW	(16-bit only) Microsoft Windows optimized prologue/epilogue code sequences (see "zW (optimized)" on page 26)
zWs	(16-bit only) Microsoft Windows smart callback sequences (see "zWs" on page 27)
zz	remove "@size" from __stdcall function names (10.0 compatible) (see "zz" on page 74)

2.2 Compiler Options - Summarized By Category

In the following sections, we present a terse summary of compiler options organized into categories.

2.2.1 Target Specific

Option:	Description:
bc	build target is a console application (see "bc" on page 20)
bd	build target is a Dynamic Link Library (DLL) (see "bd" on page 20)
bg	build target is a GUI application (see "bg" on page 20)
bm	build target is a multi-threaded environment (see "bm" on page 21)
br	build target uses DLL version of C/C++ run-time library (see "br" on page 21)
bt[=<os>]	build target for operating system <os> (see "bt[=<os>]" on page 21)
bw	build target uses default windowing support (see "bw" on page 22)
of	generate traceable stack frames as needed (see "of" on page 22)
of+	always generate traceable stack frames (see "of+" on page 23)
sg	generate calls to grow the stack (see "sg" on page 24)
st	touch stack through SS first (see "st" on page 25)
zw	generate code for Microsoft Windows (see "zw" on page 26)
zW	(16-bit only) Microsoft Windows optimized prologue/epilogue code sequences (see "zW (optimized)" on page 26)
zWs	(16-bit only) Microsoft Windows smart callback sequences (see "zWs" on page 27)

2.2.2 Debugging/Profiling

Option:	Description:
d0	(C++ only) no debugging information (see "d0" on page 28)
d1	line number debugging information (see "d1" on page 29)
d1+	(C only) line number debugging information plus typing information for global symbols and local structs and arrays (see "d1+" on page 29)
d2	full symbolic debugging information (see "d2" on page 29)
d2i	(C++ only) d2 and debug inlines; emit inlines as external out-of-line functions (see "d2i" on page 29)
d2s	(C++ only) d2 and debug inlines; emit inlines as static out-of-line functions (see "d2s" on page 29)
d2t	(C++ only) d2 but without type names (see "d2t" on page 30)
d3	full symbolic debugging with unreferenced type names (see "d3" on page 30)
d3i	(C++ only) d3 plus debug inlines; emit inlines as external out-of-line functions (see "d3i" on page 30)

d3s	(C++ only) d3 plus debug inlines; emit inlines as static out-of-line functions (see "d3s" on page 30)
ee	call epilogue hook routine (see "ee" on page 30)
en	emit routine names in the code segment (see "en" on page 30)
ep[<number>]	call prologue hook routine with number stack bytes available (see "ep[<number>]" on page 31)
et	Pentium profiling (see "et" on page 31)
h{w,d,c}	set debug output format (Watcom, Dwarf, Codeview) (see "h{w,d,c}" on page 32)
s	remove stack overflow checks (see "s" on page 32)

2.2.3 Preprocessor

Option:	***Description:***
d<name>[=text]	precompilation #define name [text] (see "d<name>[=text]" on page 32)
d+	allow extended "d" macro definitions on command line (see "d+" on page 33)
fo[=<file_name>]	set preprocessor output file name (see "fo[=<file_name>] (preprocessor)" on page 33)
p{e,l,c,w=<num>}	preprocess file

c	preserve comments
e	encrypt identifiers (C++ only)
l	insert #line directives
w=<num>	wrap output lines at <num> columns. Zero means no wrap.

(see "p{e,l,c,w=<num>}" on page 34)

u<name>	undefine macro name (see "u<name>" on page 34)

2.2.4 Diagnostics

Option:	***Description:***
e<number>	set error limit number (see "e<number>" on page 35)
ef	use full path names in error messages (see "ef" on page 35)
eq	do not display error messages (they are still written to a file) (see "eq" on page 35)
er	(C++ only) do not recover from undefined symbol errors (see "er" on page 35)

14 Compiler Options - Summarized By Category

ew	(C++ only) alternate error message formatting (see "ew" on page 35)
t=<num>	set tab stop multiplier (see "t=<num>" on page 35)
w<number>	set warning level number (see "w<number>" on page 36)
wcd=<num>	warning control: disable warning message <num> (see "wcd=<number>" on page 36)
wce=<num>	warning control: enable warning message <num> (see "wce=<number>" on page 36)
we	treat all warnings as errors (see "we" on page 36)
wx	set warning level to maximum setting (see "wx" on page 36)
z{a,e}	disable/enable language extensions (see "za" on page 36) (see "ze" on page 37)
zq	operate quietly (see "zq" on page 39)
zs	syntax check only (see "zs" on page 40)

2.2.5 Source/Output Control

Option:	***Description:***
db	generate browsing information (see "db" on page 40)
ez	generate PharLap EZ-OMF object files (see "ez" on page 40)
fc=<file_name>	(C++ only) specify file of command lines to be batch processed (see "fc=<file_name>" on page 40)
fh[q][=<file_name>]	use precompiled headers (see "fh[q][=<file_name>]" on page 41)
fhd	store debug info for pre-compiled header once (DWARF only) (see "fhd" on page 41)
fhr	(C++ only) force compiler to read pre-compiled header (will never write) (see "fhr" on page 41)
fhw	(C++ only) force compiler to write pre-compiled header (will never read) (see "fhw" on page 41)
fhwe	(C++ only) don’t include pre-compiled header warnings when "we" is used (see "fhwe" on page 41)
fi=<file_name>	force file_name to be included (see "fi=<file_name>" on page 41)
fo[=<file_name>]	set object or preprocessor output file name (see "fo[=<file_name>]" on page 41)
fr[=<file_name>]	set error file name (see "fr[=<file_name>]" on page 42)
ft	(C++ only) try truncated (8.3) header file specification (see "ft" on page 42)
fx	(C++ only) do not try truncated (8.3) header file specification (see "fx" on page 42)
i=<directory>	another include directory (see "i=<directory>" on page 43)
k	continue processing files (ignore errors) (see "k" on page 43)
v	output function declarations to .def (see "v" on page 43)

zg	generate function prototypes using base types (see "zg" on page 43)
zl	remove default library information (see "zl" on page 43)
zld	remove file dependency information (see "zld" on page 44)

2.2.6 Code Generation

Option:	*Description:*
ei	force enum base type to use at least an int (see "ei" on page 44)
em	force enum base type to use minimum (see "em" on page 44)
j	change char default from unsigned to signed (see "j" on page 44)
ri	return chars and shorts as ints (see "ri" on page 45)
xr	(C++ only) enable RTTI (see "xr" on page 45)
zc	place literal strings in the code segment (see "zc" on page 45)
zp{1,2,4,8,16}	pack structure members (see "zp[{1,2,4,8,16}]" on page 45)
zpw	output warning when padding is added in a struct/class (see "zpw" on page 48)
zt<number>	set data threshold (see "zt<number>" on page 48)
zv	(C++ only) enable virtual function removal optimization (see "zv" on page 49)

2.2.7 80x86 Floating Point

Option:	*Description:*
fpc	calls to floating-point library (see "fpc" on page 52)
fpi	in-line 80x87 instructions with emulation (see "fpi" on page 52)
fpi87	in-line 80x87 instructions (see "fpi87" on page 53)
fp2	generate floating-point for 80x87 (see "fp2" on page 54)
fp3	generate floating-point for 387 (see "fp3" on page 54)
fp5	optimize floating-point for Pentium (see "fp5" on page 54)
fp6	optimize floating-point for Pentium Pro (see "fp6" on page 54)
fpd	enable generation of Pentium FDIV bug check code (see "fpd" on page 54)

2.2.8 Segments/Modules

Option:	Description:
g=<codegroup>	set code group name (see "g=<codegroup>" on page 55)
nc=<name>	set code class name (see "nc=<name>" on page 56)
nd=<name>	set data segment name (see "nd=<name>" on page 56)
nm=<name>	set module name (see "nm=<name>" on page 57)
nt=<name>	set name of text segment (see "nt=<name>" on page 58)
zm	place each function in separate segment (near functions not allowed) (see "zm" on page 58)
zmf	(C++ only) place each function in separate segment (near functions allowed) (see "zmf" on page 59)

2.2.9 80x86 Run-time Conventions

Option:	Description:
0	(16-bit only) 8088 and 8086 instructions (see "0" on page 60)
1	(16-bit only) 188 and 186 instructions (see "1" on page 60)
2	(16-bit only) 286 instructions (see "2" on page 60)
3	(16-bit only) 386 instructions (see "3" on page 61)
4	(16-bit only) 486 instructions (see "4" on page 61)
5	(16-bit only) Pentium instructions (see "5" on page 61)
6	(16-bit only) Pentium Pro instructions (see "6" on page 61)
3r	(32-bit only) 386 register calling conventions (see "3{r|s}" on page 61)
3s	(32-bit only) 386 stack calling conventions (see "3{r|s}" on page 61)
4r	(32-bit only) 486 register calling conventions (see "4{r|s}" on page 63)
4s	(32-bit only) 486 stack calling conventions (see "4{r|s}" on page 63)
5r	(32-bit only) Pentium register calling conventions (see "5{r|s}" on page 63)
5s	(32-bit only) Pentium stack calling conventions (see "5{r|s}" on page 63)
6r	(32-bit only) Pentium Pro register calling conventions (see "6{r|s}" on page 63)
6s	(32-bit only) Pentium Pro stack calling conventions (see "6{r|s}" on page 63)
m{f,s,m,c,l,h}	memory model (Flat,Small,Medium,Compact,Large,Huge) (see "mf" on page 63)
zdf	DS floats i.e. not fixed to DGROUP (see "zd{f,p}" on page 65)
zdp	DS is pegged to DGROUP (see "zd{f,p}" on page 65)
zdl	Load DS directly from DGROUP (see "zdl" on page 65)
zff	FS floats i.e. not fixed to a segment (see "zf{f,p}" on page 65)
zfp	FS is pegged to a segment (see "zf{f,p}" on page 65)
zgf	GS floats i.e. not fixed to a segment (see "zg{f,p}" on page 65)

zgp	GS is pegged to a segment (see "zg{f,p}" on page 65)
zu	SS != DGROUP (see "zu" on page 65)

2.2.10 Optimizations

Option:	*Description:*
oa	relax aliasing constraints (see "oa" on page 66)
ob	enable branch prediction (see "ob" on page 66)
oc	disable <call followed by return> to <jump> optimization (see "oc" on page 67)
od	disable all optimizations (see "od" on page 67)
oe[=<num>]	expand user functions in-line. <num> controls max size (see "oe=<num>" on page 67)
oh	enable repeated optimizations (longer compiles) (see "oh" on page 68)
oi	expand intrinsic functions in-line (see "oi" on page 68)
oi+	(C++ only) expand intrinsic functions in-line and set inline_depth to maximum (see "oi+" on page 68)
ok	enable control flow prologues and epilogues (see "ok" on page 68)
ol	enable loop optimizations (see "ol" on page 68)
ol+	enable loop optimizations with loop unrolling (see "ol+" on page 68)
om	generate in-line 80x87 code for math functions (see "om" on page 69)
on	allow numerically unstable optimizations (see "on" on page 69)
oo	continue compilation if low on memory (see "oo" on page 69)
op	generate consistent floating-point results (see "op" on page 69)
or	reorder instructions for best pipeline usage (see "or" on page 69)
os	favor code size over execution time in optimizations (see "os" on page 69)
ot	favor execution time over code size in optimizations (see "ot" on page 70)
ou	all functions must have unique addresses (see "ou" on page 70)
ox	equivalent to -obiklmr -s (see "ox" on page 70)
oz	NULL points to valid memory in the target environment (see "oz" on page 70)

2.2.11 C++ Exception Handling

Option:	*Description:*
xd	disable exception handling (default) (see "xd" on page 71)
xdt	disable exception handling (same as "xd") (see "xdt" on page 71)
xds	disable exception handling (table-driven destructors) (see "xds" on page 71)

xs	enable exception handling (see "xs" on page 72)
xst	enable exception handling (direct calls for destruction) (see "xst" on page 72)
xss	enable exception handling (table-driven destructors) (see "xss" on page 72)

2.2.12 Double-Byte/Unicode Characters

Option:	***Description:***
zk{0,1,2,l}	double-byte char support: 0=Kanji,1=Chinese/Taiwanese,2=Korean,l=local (see "zk{0,1,2,l}" on page 72)
zk0u	translate double-byte Kanji to UNICODE (see "zk0u" on page 73)
zku=<codepage>	load UNICODE translate table for specified code page (see "zku=<codepage>" on page 73)

2.2.13 Compatibility with Microsoft Visual C++

Option:	***Description:***
vc...	VC++ compatibility options (see "vc..." on page 73)
vcap	allow alloca() or _alloca() in a parameter list

2.2.14 Compatibility with Older Versions of the 80x86 Compilers

Option:	***Description:***
r	save/restore segment registers across calls (see "r" on page 74)
fpr	generate backward compatible 80x87 code (see "fpr" on page 74)
zz	generate backward compatible __stdcall conventions by removing the "@size" from __stdcall function names (10.0 compatible) (see "zz" on page 74)

2.3 Compiler Options - Full Description

In the following sections, we present complete descriptions of compiler options organized into categories.

2.3.1 Target Specific

This group of options deals with characteristics of the target application; for example, simple executables versus Dynamic Link Libraries, character-mode versus graphical user interface, single-threaded versus multi-threaded, and so on.

bc

(OS/2, Win16/32 only) This option causes the compiler to emit into the object file references to the appropriate startup code for a character-mode console application. The presence of `LibMain/DLLMain` or `WinMain/wWinMain` in the source code does not influence the selection of startup code. Only `main` and `wmain` are significant.

If none of "bc", "bd", "bg" or "bw" are specified then the order of priority in determining which combination of startup code and libraries to use are as follows.

1. The presence of one of `LibMain` or `DLLMain` implies that the DLL startup code and libraries should be used.

2. The presence of `WinMain` or `wWinMain` implies that the GUI startup code and libraries should be used.

3. The presence of `main` or `wmain` implies that the default startup code and libraries should be used.

If both a wide and non-wide version of an entry point are specified, the "wide" entry point will be used. Thus `wWinMain` is called when both `WinMain` and `wWinMain` are present. Similarly, `wmain` is called when both `main` and `wmain` are present (and `WinMain/wWinMain` are not present). By default, if both `wmain` and `WinMain` are included in the source code, then the startup code will attempt to call `wWinMain` (since both "wide" and "windowed" entry points were included).

bd

(OS/2, Win16/32 only) This option causes the compiler to emit into the object file references to the run-time DLL startup code and, if required, special versions of the run-time libraries that support DLLs. The presence of `main/wmain` or `WinMain/wWinMain` in the source code does not influence the selection of startup code. Only `LibMain` and `DLLMain` are significant (see "bc"). The macro `__SW_BD` will be predefined if "bd" is selected.

bg

(OS/2, Win16/32 only) This option causes the compiler to emit into the object file references to the appropriate startup code for a windowed (GUI) application. The presence of

`LibMain/DLLMain` or `main/wmain` in the source code does not influence the selection of startup code. Only `WinMain` and `wWinMain` are significant (see "bc" on page 20).

bm

(Netware, OS/2, Win32 only) This option causes the compiler to emit into the object file references to the appropriate multi-threaded library name(s). The macros `_MT` and `__SW_BM` will be predefined if "bm" is selected.

br

(OS/2, Win32 only) This option causes the compiler to emit into the object file references to the run-time DLL library name(s). The run-time DLL libraries are special subsets of the Watcom C/C++ run-time libraries that are available as DLLs. When you use this option with an OS/2 application, you must also specify the "CASEEXACT" option to the Watcom Linker. The macros `_DLL` and `__SW_BR` will be predefined if "br" is selected.

bt[=<os>]

This option causes the compiler to define the "build" target. This option is used for cross-development work. It prevents the compiler from defining the default build target (which is based on the host system the compiler is running on). The default build targets are:

DOS when the host operating system is DOS,

OS2 when the host operating system is OS/2,

NT when the host operating system is Windows NT (including Windows 95), or

QNX when the host operating system is QNX.

It also prevents the compiler from defining the default target macro. Instead the compiler defines a macro consisting of the string "<os>" converted to uppercase and prefixed and suffixed with two underscores. The default target macros are described in the section entitled "Watcom C/C++ Predefined Macros" on page 89.

For example, specifying the option:

```
bt=foo
```

would cause the compiler to define the macro

```
__FOO__
```

and prevent it from defining MSDOS, _DOS and __DOS__ if the compiler was being run under DOS, __OS2__ if using the OS/2 hosted compiler, __NT__ if using the Windows NT or Windows 95 hosted compiler, or __QNX__ if using the QNX hosted version. Any string consisting of letters, digits, and the underscore character may be used for the target name.

The compiler will also construct an environment variable called **<os>_INCLUDE** and see if it has been defined. If the environment variable is defined then each directory listed in it is searched (in the order that they were specified). For example, the environment variable **WINDOWS_INCLUDE** will be searched if *bt=WINDOWS* option was specified.

Example:

```
set windows_include=\watcom\h\win
```

Include file processing is described in the section entitled "Watcom C/C++ #include File Processing" on page 85.

Several target names are recognized by the compiler and perform additional operations.

Target name	***Additional operation***
DOS	Defines the macros _DOS and MSDOS.
WINDOWS	Same as specifying one of the "zw" options. Defines the macros _WINDOWS (16-bit only) and __WINDOWS_386__ (32-bit only).
NETWARE	(32-bit only) Causes the compiler to use stack-based calling conventions. Also defines the macro __NETWARE_386__.

Specifying "bt" with no target name following restores the default target name.

bw

(OS/2, Win16, Win32 only) This option causes the compiler to import a special symbol so that the default windowing library code is linked into your application. The presence of LibMain/DLLMain in the source code does not influence the selection of startup code. Only main, wmain, WinMain and wWinMain are significant (see "bc" on page 20). The macro __SW_BW will be predefined if "bw" is selected.

of

This option selects the generation of traceable stack frames for those functions that contain calls or require stack frame setup.

(16-bit only) To use Watcom's "Dynamic Overlay Manager" (DOS only), you must compile all modules using one of the "of" or "of+" options ("of" is sufficient).

For near functions, the following function prologue sequence is generated.

```
(16-bit only)
    push BP
    mov  BP,SP

(32-bit only)
    push EBP
    mov  EBP,ESP
```

For far functions, the following function prologue sequence is generated.

```
(16-bit only)
    inc  BP
    push BP
    mov  BP,SP

(32-bit only)
    inc  EBP
    push EBP
    mov  EBP,ESP
```

The BP/EBP value on the stack will be even or odd depending on the code model.

For 16-bit DOS systems, the Dynamic Overlay Manager uses this information to determine if the return address on the stack is a short address (16-bit offset) or long address (32-bit segment:offset).

Do not use this option for 16-bit Windows applications. It will alter the code sequence generated for "_export" functions.

Example:

```
C>compiler_name toaster /of
```

The macro `__SW_OF` will be predefined if "of" is selected.

of+

This option selects the generation of traceable stack frames for all functions regardless of whether they contain calls or require stack frame setup. This option is intended for developers of embedded systems (ROM-based applications).

To use Watcom's "Dynamic Overlay Manager" (16-bit DOS only), you must compile all modules using one of the "of" or "of+" options ("of" is sufficient).

For near functions, the following function prologue sequence is generated.

```
(16-bit only)
    push BP
    mov  BP,SP

(32-bit only)
    push EBP
    mov  EBP,ESP
```

For far functions, the following function prologue sequence is generated.

```
(16-bit only)
    inc  BP
    push BP
    mov  BP,SP

(32-bit only)
    inc  EBP
    push EBP
    mov  EBP,ESP
```

The BP/EBP value on the stack will be even or odd depending on the code model.

For 16-bit DOS systems, the Dynamic Overlay Manager uses this information to determine if the return address on the stack is a short address (16-bit offset) or long address (32-bit segment:offset).

Do not use this option for 16-bit Windows applications. It will alter the code sequence generated for "_export" functions.

Example:

```
C>compiler_name toaster /of+
```

sg

This option is useful for 32-bit OS/2 multi-threaded applications. It requests the code generator to emit a run-time call at the start of any function that has more than 4K bytes of automatic variables (variables located on the stack).

Under 32-bit OS/2, the stack is grown automatically in 4K pages for any threads, other than the primary thread, using the stack "guard page" mechanism. The stack consists of in-use committed pages topped off with a special guard page. A memory reference into the 4K

guard page causes the operating system to grow the stack by one 4K page and to add a new 4K guard page. This works fine when there is less than 4K of automatic variables in a function. When there is more than 4K of automatic data, the stack must be grown in an orderly fashion, 4K bytes at a time, until the stack has grown sufficiently to accommodate all the automatic variable storage requirements. Hence the requirement for a stack-growing run-time routine. The stack-growing run-time routine is called `__GRO.`

The "stack=" linker option specifies how much stack is available and committed for the primary thread when an executable starts. The stack size parameter to `_beginthread()` specifies how much stack is available for a child thread. The child thread starts with just 4k of stack committed. The stack will not grow to be bigger than the size specified by the stack size parameter.

Under 32-bit Windows (Win32), the stack is grown automatically in 4K pages for all threads using a similar stack "guard page" mechanism. The stack consists of in-use committed pages topped off with a special guard page. The techniques for growing the stack in an orderly fashion are the same as that described above for OS/2.

The "stack=" linker option specifies how much stack is available for the primary thread when an executable starts. The "commit stack=" linker directive specifies how much of that stack is committed when the executable starts. If no "commit stack=" directive is used, it defaults to the same value as the stack size. The stack size parameter to `_beginthread()` specifies how much stack is committed for a child thread. If the size is set to zero, the size of the primary thread stack is used for the child thread stack. When the child thread executes, the stack space is not otherwise restricted.

The macro `__SW_SG` will be predefined if "sg" is selected.

st

This option causes the code generator to ensure that the first reference to the stack in a function is to the stack "bottom" using the SS register. If the memory for this part of the stack is not mapped to the task, a memory fault will occur involving the SS register. This permits an operating system to allocate additional stack space to the faulting task.

Suppose that a function requires 100 bytes of stack space. The code generator usually emits an instruction sequence to reduce the stack pointer by the required number of bytes of stack space, thereby establishing a new stack bottom. When the "st" option is specified, the code generator will ensure that the first reference to the stack is to a memory location with the lowest address. If a memory fault occurs, the operating system can determine that it was a stack reference (since the SS register is involved) and also how much additional stack space is required.

See the description of the "sg" option for a more general solution to the stack allocation problem. The macro __SW_ST will be predefined if "st" is selected.

zw

(16-bit only) This option causes the compiler to generate the prologue/epilogue code sequences required for Microsoft Windows applications. The following "fat" prologue/epilogue sequence is generated for any functions declared to be "far _export" or "far pascal".

```
far pascal func(...)
far _export func(...)
far _export pascal func(...)

    push DS
    pop  AX
    nop
    inc  BP
    push BP
    mov  BP,SP
    push DS
    mov  DS,AX
    .
    .
    .
    pop DS
    pop BP
    dec BP
    retf n
```

The macro __WINDOWS__ will be predefined if "zw" is selected.

(32-bit only) This option causes the compiler to generate any special code sequences required for 32-bit Microsoft Windows applications. The macro __WINDOWS__ and __WINDOWS_386__ will be predefined if "zw" is selected.

zW (optimized)

(16-bit only) This option is similar to "zw" but causes the compiler to generate more efficient prologue/epilogue code sequences in some cases. This option may be used for Microsoft Windows applications code other than user callback functions. Any functions declared as "far _export" will be compiled with the "fat" prologue/epilogue code sequence described under the "zw" option.

```
far _export func(...)
far _export pascal func(...)
```

26 Compiler Options - Full Description

The following "skinny" prologue/epilogue sequence is generated for functions that are not declared to be "far _export".

```
far pascal func(...)
far func(...)

    inc  BP
    push BP
    mov  BP,SP
    .
    .
    .
    pop BP
    dec BP
    retf n
```

The macro __WINDOWS__ will be predefined if "zW" is selected.

zWs

(16-bit only) This option is similar to "zW" but causes the compiler to generate "smart callbacks". This option may be used for Microsoft Windows user callback functions in executables only. It is not permitted for DLLs. Normally, a callback function cannot be called directly. You must use MakeProcInstance to obtain a function pointer with which to call the callback function.

If you specify "zWs" then you do not need to use MakeProcInstance in order to call your own callback functions. Any functions declared as "far _export" will be compiled with the "smart" prologue code sequence described here.

The following example shows the usual prologue code sequence that is generated when the "zWs" option is NOT used.

Example:

compiler_name `winapp /mc /bt=windows /d1`

```
short FAR PASCAL __export Function1( short var1,
                                     long varlong,
                                     short var2 )
{
 0000  1e            FUNCTION1       push    ds
 0001  58                            pop     ax
 0002  90                            nop
 0003  45                            inc     bp
 0004  55                            push    bp
 0005  89 e5                         mov     bp,sp
 0007  1e                            push    ds
 0008  8e d8                         mov     ds,ax
```

The following example shows the "smart" prologue code sequence that is generated when the "zWs" option is used. The assumption here is that the SS register contains the address of DGROUP.

Example:

compiler_name `winapp /mc /bt=windows /d1 /zWs`

```
short FAR PASCAL __export Function1( short var1,
                                     long varlong,
                                     short var2 )
{
 0000  8c d0         FUNCTION1       mov     ax,ss
 0002  45                            inc     bp
 0003  55                            push    bp
 0004  89 e5                         mov     bp,sp
 0006  1e                            push    ds
 0007  8e d8                         mov     ds,ax
```

2.3.2 Debugging/Profiling

This group of options deals with all the forms of debugging information that can be included in an object file. Support for profiling of Pentium code is also described.

d0

(C++ only) No debugging information is included in the object file.

d1

Line number debugging information is included in the object file. This option provides additional information to the Watcom Debugger (at the expense of larger object files and executable files). Line numbers are handy when debugging your application with the Watcom Debugger at the source code level. Code speed is not affected by this option. To avoid recompiling, the Watcom Strip Utility can be used to remove debugging information from the executable image.

d1+

(C only) Line number debugging information plus typing information for global symbols and local structs and arrays is included in the object file. Although global symbol information can be made available to the Watcom Debugger through a Watcom Linker option, typing information for global symbols and local structs and arrays must be requested when the source file is compiled. This option provides additional information to the Watcom Debugger (at the expense of larger object files and executable files). Code speed is not affected by this option. To avoid recompiling, the Watcom Strip Utility can be used to remove debugging information from the executable image.

d2

In addition to line number information, local symbol and data type information is included in the object file. Although global symbol information can be made available to the Watcom Debugger through a Watcom Linker option, local symbol and typing information must be requested when the source file is compiled. This option provides additional information to the Watcom Debugger (at the expense of larger object files and executable files).

By default, the compiler will select the "od" level of optimization if "d2" is specified (see the description of the "od" option). Starting with version 11, the compiler now expands functions in-line where appropriate. This means that symbolic information for the in-lined function will not be available.

The use of this option will make the debugging chore somewhat easier at the expense of code speed and size. To create production code, you should recompile without this option.

d2i

(C++ only) This option is identical to "d2" but does not permit in-lining of functions. Functions are emitted as external out-of-line functions. This option can result in larger object and/or executable files than with "d2" (we are discussing both "code" and "file" size here).

d2s

(C++ only) This option is identical to "d2" but does not permit in-lining of functions. Functions are emitted as static out-of-line functions. This option can result in larger object

and/or executable files than with "d2" or "d2i" (we are discussing both "code" and "file" size here). Link times are faster than "d2i" (fewer segment relocations) but executables are slightly larger.

d2t

(C++ only) This option is identical to "d2" but does not include type name debugging information. This option can result in smaller object and/or executable files (we are discussing "file" size here).

d3

This option is identical to "d2" but also includes symbolic debugging information for unreferenced type names. Note that this can result in very large object and/or executable files when header files like `WINDOWS.H` or `OS2.H` are included.

d3i

(C++ only) This option is identical to "d3" but does not permit in-lining of functions. Functions are emitted as external out-of-line functions. This option can result in larger object and/or executable files than with "d3" (we are discussing both "code" and "file" size here).

d3s

(C++ only) This option is identical to "d3" but does not permit in-lining of functions. Functions are emitted as static out-of-line functions. This option can result in larger object and/or executable files than with "d3" or "d3i" (we are discussing both "code" and "file" size here). Link times are faster than "d3i" (fewer segment relocations) but executables are slightly larger.

ee

This option causes the compiler to generate a call to `__EPI` in the epilogue sequence at the end of every function. This user-written routine can be used to collect/record profiling information. Other related options are "ep[<number>]" on page 31 and "en". The macro `__SW_EE` will be predefined if "ee" is selected.

en

The compiler will emit the function name into the object code as a string of characters just before the function prologue sequence is generated. The string is terminated by a byte count of the number of characters in the string.

```
; void Toaster( int arg )

        db      "Toaster", 7
public  Toaster
Toaster label   byte
        .
        .
        .
        ret
```

This option is intended for developers of embedded systems (ROM-based applications). It may also be used in conjunction with the "ep" option for special user-written profiling applications. The macro __SW_EN will be predefined if "en" is selected.

ep[<number>]

This option causes the compiler to generate a call to a user-written __PRO routine in the prologue sequence at the start of every function. This routine can be used to collect/record profiling information. The optional argument <*number*> can be used to cause the compiler to allocate that many bytes on the stack as a place for __PRO to store information. Other related options are "ee" on page 30 and "en" on page 30. The macro __SW_EP will be predefined if "ep" is selected.

et

(Pentium only) This option causes the compiler to generate code into the prolog of each function to count exactly how much time is spent within that function, in clock ticks. This option is valid only for Pentium compatible processors (i.e., the instructions inserted into the code do not work on 486 or earlier architectures). The Pentium "rdtsc" opcode is used to obtain the instruction cycle count.

At the end of the execution of the program, a file will be written to the same location as the executable, except with a ".prf" extension. The contents of the file will look like this:

Example:

```
1903894223          1  main
1785232334    1376153  StageA
1882249150      13293  StageB
1830895850       2380  StageC
 225730118         99  StageD
```

The first column is the total number of clock ticks spent inside of the function during the execution of the program, the second column is the number of times it was called and the third column is the individual function name. The total number of clock ticks includes time spent within functions called from this function.

The overhead of the profiling can be somewhat intrusive, especially for small leaf functions (i.e., it may skew your results somewhat).

h{w,d,c}

The type of debugging information that is to be included in the object file is one of "Watcom", "Dwarf" or "Codeview". The default is "Dwarf".

If you wish to use the Microsoft Codeview debugger, then choose the "hc" option (this option causes Codeview Level 4 information to be generated). It will be necessary to run the Microsoft Debugging Information Compactor, CVPACK, on the executable once the linker has created it. For information on requesting the linker to automatically run CVPACK, see the section entitled "OPTION CVPACK" in the ***Watcom Linker User's Guide.*** Alternatively, you can run CVPACK from the command line.

When linking the application, you must also choose the appropriate Watcom Linker DEBUG directive. See the ***Watcom Linker User's Guide*** for more information.

s

Stack overflow checking is omitted from the generated code. By default, the compiler will emit code at the beginning of every function that checks for the "stack overflow" condition. This option can be used to disable this feature. The macro `__SW_S` will be predefined if "s" is selected.

2.3.3 Preprocessor

This group of options deals with the compiler preprocessor.

d<name>[=text]

This option can be used to define a preprocessor macro from the command line. If *=text* is not specified, then 1 is assumed. In other words, specifying `/dDBGON` is equivalent to specifying `/dDBGON=1` on the command line.

If *=text* is specified, then this option is equivalent to including the following line in your source code.

```
#define name text
```

Consider the following example.

Example:

```
d_MODDATE="87.05.04"
```

The above example is equivalent to a line in the source file containing:

```
#define _MODDATE "87.05.04"
```

d+

The syntax of any "d" option which follows on the command line is extended to include C/C++ tokens as part of "text". The token string is terminated by a space character. This permits more complex syntax than is normally allowed.

Example:

```
/d+ /d_radx=x*3.1415926/180
```

This is equivalent to specifying the following in the source code.

Example:

```
#define _radx x*3.1415926/180
```

Watcom C++ extends this feature by allowing parameterized macros. When a parameter list is specified, the "=" character must not be specified. It also permits immediate definition of the macro as shown in the second line of the example.

Example:

```
/d+ /d_rad(x)x*3.1415926/180
/d+_rad(x)x*3.1415926/180
```

This is equivalent to specifying the following in the source code.

Example:

```
#define _rad(x) x*3.1415926/180
```

fo[=<file_name>] (preprocessor)

The "fo" option is used with any form of the "p" (preprocessor) option to name the output file drive, path, file name and extension. If the output file name is not specified, it is constructed from the source file name. If the output file extension is not specified, it is ".i" by default.

Example:

C>*compiler_name* `report /p /fo=d:\proj\prep\`

A trailing "\" must be specified for directory names. If, for example, the option was specified as `fo=d:\proj\prep` then the output file would be called `D:\PROJ\PREP.I`. A default filename extension must be preceded by a period (".").

Example:

```
C>compiler_name report /p /fo=d:\proj\prep\.cpr
```

p{e,l,c,w=<num>}

The input file is preprocessed and, by default, is written to the standard output file. The "fo" option may be used to redirect the output to a file with default extension ".i".

Specify "pc" if you wish to include the original source comments in the Watcom C/C++ preprocessor output file.

(C++ Only) Specify "pe" if you wish to encrypt the original identifiers when they are written to the Watcom C/C++ preprocessor output file.

Specify "pl" if you wish to include `#line` directives.

Specify "pcl" or "plc" if you wish both source comments and #line directives.

Use the "w=<num>" suffix if you wish to wish output lines to wrap at <num> columns. Zero means no wrap.

Example:

```
C>compiler_name report /pcelw=80
```

The input file is preprocessed only. When only "p" is specified, source comments and #line directives are not included. You must request these using the "c" and "l" suffixes. When the output of the preprocessor is fed into the compiler, the `#line` directive enables the compiler to issue diagnostics in terms of line numbers of the original source file.

The options which are supported when the Watcom C/C++ preprocessor is requested are: "d", "fi", "fo", "i", "m?", and "u".

u<name>

The "u" option may be used to turn off the definition of a predefined macro. If no name is specified then all predefined macros are undefined.

Example:

```
C>compiler_name report /uM_I386
```

2.3.4 Diagnostics

This group of options deals with the control of compiler diagnostics.

e<number>

The compiler will stop compilation after reaching <*number*> errors. By default, the compiler will stop compilation after 20 errors.

ef

This option causes the compiler to display full path names for files in error messages.

eq

This option causes the compiler to not display error messages on the console; however, they are still written to a file (see "fr[=<file_name>]" on page 42).

er

(C++ only) This option causes the C++ compiler to not recover from undefined symbol errors. By default, the compiler recovers from "undefined symbol" errors by injecting a special entry into the symbol table that prevents further issuance of diagnostics relating to the use of the same name. Specify the "er" option if you want all uses of the symbol to be diagnosed.

Example:

```
struct S {
};

void foo( S *p ) {
    p->m = 1; // member 'm' has not been declared in 'S'

}
void bar( S *p ) {
    p->m = 2; // no error unless "er" is specified
}
```

ew

(C++ only) This option causes the C++ compiler to generate equivalent but less verbose diagnostic messages.

t=<num>

(C++ only) The "t" option is used to set the tab stop interval. By default, the compiler assumes a tab stop occurs at multiples of 8 (1+n x 8 = 1, 9, 17, ... for n=0, 1, 2, ...). When the

compiler reports a line number and column number in a diagnostic message, the column number has been adjusted for intervening tabs. If the default tab stop setting for your text editor is not a multiple of 8, then you should use this option.

Example:

```
C>compiler_name report /t=4
```

w<number>

The compiler will issue only warning type messages of severity <*number*> or below. Type 1 warning messages are the most severe while type 3 warning messages are the least severe. Specify "w0" to prevent warning messages from being issued. Specify "wx" to obtain all warning messages.

wcd=<number>

The compiler will not issue the warning message indicated by <*number*>.

wce=<number>

The compiler will issue the warning message indicated by <*number*> despite any pragmas that may have disabled it.

we

By default, the compiler will continue to create an object file when there are warnings produced. This option can be used to treat all warnings as errors, thereby preventing the compiler from creating an object file if there are warnings found within a module.

wo

(C only) (16-bit only) This option tells the compiler to emit warnings for things that will cause problems when compiling code for use in overlays.

wx

This option sets the warning level to its maximum setting.

za

This option helps to ensure that the module to be compiled conforms to the ANSI C or C++ programming language specification (depending on the compiler which is selected). The macro `NO_EXT_KEYS` (no extended keywords) will be predefined if "za" is selected. The "ou" option will be enabled (see "ou" on page 70). See also the description of the "ze" option.

When using the C compiler, there is an exception to the enforcement of the ANSI C standard programming language specification. The use of C++ style comments (// comment) are not diagnosed.

ze

The "ze" option (default) enables the use of the following compiler extensions:

1. The requirement for at least one external definition per module is relaxed.

2. When using the C compiler, some forgiveable pointer type mismatches become warnings instead of errors.

3. In-line math functions are allowed (note that *errno* will not be set by in-line functions).

4. When using the C compiler, anonymous structs/unions are allowed (this is always permitted in C++).

   Example:

   ```
   struct {
       int a;
       union {
           int   b;
           float alt_b;
       };
       int c;
   } x;
   ```

   In the above example, "x.b" is a valid reference to the "b" field.

5. For C only, ANSI function prototype scope rules are relaxed to allow the following program to compile without any errors.

Example:

```
void foo( struct a *__p );

struct a {
    int b;
    int c;
};

void bar( void )
{
    struct a x;
    foo( &x );
}
```

According to a strict interpretation of the ANSI C standard, the function prototype introduces a new scope which is terminated at the semicolon (;). The effect of this is that the structure tag "a" in the function "foo" is not the same structure tag "a" defined after the prototype. A diagnostic must be issued for a conforming ANSI C implementation.

6. A trailing comma (,) is allowed after the last constant in an enum declaration.

   Example:

   ```
   enum colour { RED, GREEN, BLUE, };
   ```

7. The ANSI requirement that all enums have a base type of *int* is relaxed. The motivation for this extension is conservation of storage. Many enums can be represented by integral types that are smaller in size than an *int.*

   Example:

   ```
   enum colour { RED, GREEN, BLUE, };

   void foo( void )
   {
       enum colour x;

       x = RED;
   }
   ```

   In the example, "x" can be stored in an *unsigned char* because its values span the range 0 to 2.

8. The ANSI requirement that the base type of a bitfield be *int* or *unsigned* is relaxed. This allows a programmer to allocate bitfields from smaller units of storage than an *int* (e.g., *unsigned char*).

Example:

```
struct {
    unsigned char a : 1;
    unsigned char b : 1;
    unsigned char c : 1;
} x;

struct {
    unsigned a : 1;
    unsigned b : 1;
    unsigned c : 1;
} y;
```

In the above example, the size of "x" is the same size as an *unsigned char* whereas the size of "y" is the same size as an *unsigned int.*

9. The following macros are defined.

```
_near, near
_far, far, SOMDLINK (16-bit)
_huge, huge
_based
_segment
_segname
_self
_cdecl, cdecl, _Cdecl, SOMLINK (16-bit)
_pascal, pascal, _Pascal
_fortran, fortran
_interrupt, interrupt
_export
_loadds
_saveregs
_syscall, _System, SOMLINK (32-bit), SOMDLINK
(32-bit)
_far16, _Far16
```

See also the description of the "za" option.

zq

The "quiet mode" option causes the informational messages displayed by the compiler to be suppressed. Normally, messages are displayed identifying the compiler and summarizing the number of lines compiled. As well, a dot is displayed every few seconds while the code generator is active, to indicate that the compiler is still working. These messages are all suppressed by the "quiet mode" option. Error and warning messages are not suppressed.

zs

The compiler will check the source code only and omit the generation of object code. Syntax checking, type checking, and so on are performed as usual.

2.3.5 Source/Output Control

This group of options deals with control over the input files and output files that the compiler processes and/or creates.

db

Use this option to generate browsing information. The browsing information is recorded in a file whose name is constructed from the source file name and the extension ".mbr".

ez

(32-bit only) The compiler will generate an object file in Phar Lap Easy OMF-386 (object module format) instead of the default Microsoft OMF. The macro `__SW_EZ` will be predefined if "ez" is selected.

fc=<file_name>

(C++ only) The specified "batch" file contains a list of command lines to be processed. This enables the processing of a number of source files together with options for each file with one single invocation of the compiler. Only one "fc" option is allowed and no source file names are permitted on the command line.

Example:

```
[batch.txt]
main        /oneatx /zp4
part1 part2 /oneatx /zp4 /d1
part3       /oneatx /zp4 /d2

C>compiler_name /fc=\watcom\h\batch.txt
```

Each line in the file is treated stand-alone. In other words, the options from one line do not carry over to another line. However, any options specified on the command line or the associated compiler environment variable will carry over to the individual command lines in the batch file. When the compiler diagnoses errors in a source file, processing of subsequent command lines is halted unless the "k" option was specified (see "k" on page 43).

fh[q][=<file_name>]

The compiler will generate/use a precompiled header for the first header file referenced by `#include` in the source file. See the chapter entitled "Precompiled Headers" on page 117 for more information.

fhd

The compiler will store debug info for the pre-compiled header once (DWARF only). See the chapter entitled "Precompiled Headers" on page 117 for more information.

fhr

(C++ only) This option will force the compiler to read the pre-compiled header if it appears to be up-to-date; otherwise, it will read the header files included by the source code. It will never write the pre-compiled header (even when it is out-of-date). See the chapter entitled "Precompiled Headers" on page 117 for more information.

fhw

(C++ only) This option will force the compiler to write the pre-compiled header (even when it appears to be up-to-date). See the chapter entitled "Precompiled Headers" on page 117 for more information.

fhwe

(C++ only) This option will ensure that pre-compiled header warnings are not counted as errors when the "we" (treat warnings as errors) option is specified.

fi=<file_name>

The specified file is included as if a

```
#include "<file_name>"
```

directive were placed at the start of the source file.

Example:

```
C>compiler_name report /fi=\watcom\h\stdarg.h
```

fo[=<file_name>]

When generating an object file, the "fo" option may be used to name the object file drive, path, file name and extension. If the object file name is not specified, it is constructed from the source file name. If the object file extension is not specified, it is ".obj" by default.

Example:

```
C>compiler_name report /fo=d:\proj\obj\
```

A trailing "\" must be specified for directory names. If, for example, the option was specified as `fo=d:\proj\obj` then the object file would be called `D:\PROJ\OBJ.OBJ`.

A default filename extension must be preceded by a period (".").

Example:

```
C>compiler_name report /fo=d:\proj\obj\.dbo
```

fr[=<file_name>]

The "fr" option is used to name the error file drive, path, file name and extension. If the error file name is not specified, it is constructed from the source file name. If the output file extension is not specified, it is ".err" by default. If no part of the name is specified, then no error file is produced (i.e., /fr disables production of an error file).

Example:

```
C>compiler_name report /fr=d:\proj\errs\
```

A trailing "\" must be specified for directory names. If, for example, the option was specified as `fr=d:\proj\errs` then the output file would be called `D:\PROJ\ERRS.ERR`. A default filename extension must be preceded by a period (".").

Example:

```
C>compiler_name report /fr=d:\proj\errs\.erf
```

ft

(C++ only) If the compiler cannot open a header file whose file name is longer than 8 letters or whose file extension is longer than 3 letters, it will truncate the name at 8 letters and the extension at 3 letters and try to open a file with the shortened name. This is the default behaviour for the compiler.

For example, if the compiler cannot open the header file called `STRSTREAM.H`, it will attempt to open a header file called `STRSTREA.H`.

fx

(C++ only) This option can be used to disable the truncated header filename processing that the compiler does by default (see "ft" above).

i=<directory>

where "<directory>" takes the form

```
[d:]path;[d:]path...
```

The specified paths are added to the list of directories in which the compiler will search for "include" files. See the section entitled "Watcom C/C++ #include File Processing" on page 85 for information on directory searching.

k

(C++ only) This option instructs the compiler to continue processing subsequent source files after an error has been diagnosed in the current source file. See the option "fc=<file_name>" on page 40 for information on compiling multiple source files.

v

Watcom C will output function declarations to a file with the same filename as the C source file but with extension ".def". The "definitions" file may be used as an "include" file when compiling other modules in order to take advantage of the compiler's function and argument type checking.

zg

The "zg" option is similar to the "v" option except that function declarations will be output to the "DEF" file using base types (i.e., typedefs are reduced to their base type).

Example:

```
typedef unsigned int UINT;
UINT f( UINT x )
 {
    return( x + 1 );
 }
```

If you use the "v" option, the output will be:

```
extern UINT f(UINT );
```

If you use the "zg" option, the output will be:

```
extern unsigned int f(unsigned int );
```

zl

By default, the compiler places in the object file the names of the C libraries that correspond to the memory model and floating-point options that were selected. The Watcom Linker uses

these library names to select the libraries required to link the application. If you use the "zl" option, the library names will not be included in the generated object file.

The compiler may generate external references for library code that conveniently cause the linker to link in different code. One such case is: if you have any functions that pass or return floating-point values (i.e., float or double), the compiler will insert an external reference that will cause the floating-point formatting routines to be included in the executable. The "zl" option will disable these external references.

Use this option when you wish to create a library of object modules which do not contain Watcom C/C++ library name references.

zld

By default, the compiler places in the object file the names and time stamps of all the files referenced by the source file. This file dependency information can then be used by WMAKE to determine that this file needs to be recompiled if any of the referenced files has been modified since the object file was created. This option causes the compiler to not emit this information into the object file.

2.3.6 Code Generation

This group of options deals with controlling some aspects of the code that is generated by the compiler.

ei

This option can be used to force the compiler to allocate at least an "int" for all enumerated types. The macro `__SW_EI` will be predefined if "ei" is selected.

em

This option can be used to force the compiler to allocate the smallest storage unit required to hold all possible values given for an enumerated list. This option is the default for the x86 architecture. The macro `__SW_EM` will be predefined if "em" is selected.

j

The default `char` type is changed from an unsigned to a signed quantity. The macros `__CHAR_SIGNED__` and `__SW_J` will be predefined if "j" is selected.

ri

Functions declared to return integral types such as chars and shorts are promoted to returning ints. This allows non-ANSI-conforming source code which does not properly declare the return types of functions to work properly. The use of this option should be avoided.

xr

The "xr" option is used to to enable the use of the C++ feature called Run-Time Type Information (RTTI). RTTI can only be used with classes that have virtual functions declared. This restriction implies that if you enable RTTI, the amount of storage used for a class in memory does not change. The RTTI information is added to the virtual function information block so there will be an increase in the executable size if you choose to enable RTTI. There is no execution penalty at all unless you use the dynamic_cast<> feature in which case, you should be aware that the operation requires a lookup operation in order to perform the conversion properly. You can mix and match modules compiled with and without "xr", with the caveat that dynamic_cast<> and typeid() may not function (return NULL or throw an exception) if used on a class instance that was not compiled with the "xr" option.

zc

The "zc" option causes the code generator to place literal strings and *const* items in the code segment.

Example:

```
extern const int cvar = 1;
int var = 2;
const int ctable[ 5 ] = { 1, 2, 3, 4, 5 };
char *birds[ 3 ] = { "robin", "finch", "wren" };
```

In the above example, `cvar` and `ctable` and the strings `"robin", "finch",` etc. are placed in the code segment. This option is supported in large data or flat memory models only, or if the item is explicitly "far". The macro `__SW_ZC` will be predefined if "zc" is selected.

zp[{1,2,4,8,16}]

The "zp" option allows you to specify the alignment of members in a structure. The default is "zp2" for the 16-bit compiler and "zp8" for 32-bit compiler. The alignment of structure members is described in the following table. If the size of the member is 1, 2, 4, 8 or 16, the alignment is given for each of the "zp" options. If the member of the structure is an array or structure, the alignment is described by the row "x".

sizeof(member)	zp1	zp2	zp4	zp8	zp16
1	0	0	0	0	0
2	0	2	2	2	2
4	0	2	4	4	4
8	0	2	4	8	8
16	0	2	4	8	16
x	aligned to largest member				

An alignment of 0 means no alignment, 2 means word boundary, 4 means doubleword boundary, etc.

Note that packed structures are padded to ensure that consecutive occurrences of the same structure in memory are aligned appropriately. This is illustrated when the following example is compiled with "zp4". The amount of padding is determined as follows. If the largest member of structure "s" is 1 byte then "s" is not aligned. If the largest member of structure "s" is 2 bytes then "s" is aligned according to row 2. If the largest member of structure "s" is 4 bytes then "s" is aligned according to row 4. If the largest member of structure "s" is 8 bytes then "s" is aligned according to row 8. At present, there are no scalar objects that can have a size of 16 bytes. If the largest member of structure "s" is an array or structure then "s" is aligned according to row "x". Padding is the inclusion of slack bytes at the end of a structure in order to guarantee the alignment of consecutive occurrences of the same structure in memory.

To understand why structure member alignment may be important, consider the following example.

Example:

```
#include <stdio.h>
#include <stddef.h>

typedef struct memo_el {
    char           date[9];
    struct memo_el *prev,*next;
    int            ref_number;
    char           sex;
} memo;
```

```
void main( )
{
    printf( "Offset of %s is %d\n",
            "date", offsetof( memo, date ) );
    printf( "Offset of %s is %d\n",
            "prev", offsetof( memo, prev ) );
    printf( "Offset of %s is %d\n",
            "next", offsetof( memo, next ) );
    printf( "Offset of %s is %d\n",
            "ref_number", offsetof( memo, ref_number ) );
    printf( "Offset of %s is %d\n",
            "sex", offsetof( memo, sex ) );
    printf( "Size of %s is %d\n",
            "memo", sizeof( memo ) );
    printf( "Number of padding bytes is %d\n",
            sizeof( memo )
            - (offsetof( memo, sex ) + sizeof( char )) );
}
```

In the above example, the default alignment "zp8" will cause the pointer and integer items to be aligned on even addresses although the array "date" is 9 bytes in length. The items are 2-byte aligned when sizeof(item) is 2 and 4-byte aligned when sizeof(item) is 4. On computer systems that have a 16-bit (or 32-bit) bus, improved performance can be obtained when pointer, integer and floating-point items are aligned on an even boundary. This could be done by careful rearrangement of the fields of the structure or it can be forced by use of the "zp" option.

```
16-bit output when compiled zp1:
Offset of date is 0
Offset of prev is 9
Offset of next is 11
Offset of ref_number is 13
Offset of sex is 15
Size of memo is 16
Number of padding bytes is 0

16-bit output when compiled zp4:
Offset of date is 0
Offset of prev is 10
Offset of next is 12
Offset of ref_number is 14
Offset of sex is 16
Size of memo is 18
Number of padding bytes is 1
```

```
32-bit output when compiled zp1:
Offset of date is 0
Offset of prev is 9
Offset of next is 13
Offset of ref_number is 17
Offset of sex is 21
Size of memo is 22
Number of padding bytes is 0

32-bit output when compiled zp4:
Offset of date is 0
Offset of prev is 12
Offset of next is 16
Offset of ref_number is 20
Offset of sex is 24
Size of memo is 28
Number of padding bytes is 3
```

zpw

The compiler will output a warning message whenever padding is added to a struct/class for alignment purposes.

zt<number>

The "data threshold" option is used to set the maximum size for data objects to be included in the default data segment. This option can be used with the compact, large, and huge (16-bit) memory models only. These are memory models where there can be more than one data segment. Normally, all data objects whose size is less than or equal to the threshold value are placed in the default data segment "_DATA" unless they are specifically declared to be `far` items. When there is a large amount of static data, it is often useful to set the data threshold size so that all objects larger than this size are placed in another (far) data segment. For example, the option "zt100" causes all data objects larger than 100 bytes in size to be implicitly declared as `far` and grouped in other data segments.

The default data threshold value is 32767. Thus, by default, all objects greater than 32767 bytes in size are implicitly declared as `far` and will be placed in other data segments. If the "zt" option is specified without a size, the data threshold value is 256. The largest value that can be specified is 32767 (a larger value will result in 256 being selected).

If the "zt" option is used to compile any module in a program, then you must compile all the other modules in the program with the same option (and value).

Care must be exercised when declaring the size of objects in different modules. Consider the following declarations in two different C files. Suppose we define an array in one module as follows:

```
extern int Array[100] = { 0 };
```

and, suppose we reference the same array in another module as follows:

```
extern int Array[10];
```

Assuming that these modules were compiled with the option "zt100", we would have a problem. In the first module, the array would be placed in another segment since `Array[100]` is bigger than the data threshold. In the second module, the array would be placed in the default data segment since `Array[10]` is smaller than the data threshold. The extra code required to reference the object in another data segment would not be generated.

Note that this problem can also occur even when the "zt" option is not used (i.e., for objects greater than 32767 bytes in size). There are two solutions to this problem: (1) be consistent when declaring an object's size, or, (2) do not specify the size in data reference declarations.

zv

(C++ only) Enable virtual function removal optimization.

2.3.7 80x86 Floating Point

This group of options deals with control over the type of floating-point instructions that the compiler generates. There are two basic types — floating-point calls (FPC) or floating-point instructions (FPI). They are selectable through the use of one of the compiler options described below. You may wish to use the following list when deciding which option best suits your requirements. Here is a summary of advantages/disadvantages to both.

FPC

1. not IEEE floating-point
2. not tailorable to processor
3. uses coprocessor if present; simulates otherwise
4. 32-bit/64-bit accuracy
5. runs somewhat faster if coprocessor present
6. faster emulation (fewer bits of accuracy)
7. leaner "math" library
8. fatter application code (calls to library rather than in-line instructions)
9. application cannot trap floating-point exceptions
10. ideal for ROM applications

FPI, FPI87

1. IEEE floating-point
2. tailorable to processor (see fp2, fp3, fp5, fp6)
3. uses coprocessor if present; emulates IEEE otherwise
4. up to 80-bit accuracy
5. runs "full-tilt" if coprocessor present
6. slower emulation (more bits of accuracy)
7. fatter "math" library
8. leaner application code (in-line instructions)
9. application can trap floating-point exceptions
10. ideal for general-purpose applications

To see the difference in the type of code generated, consider the following small example.

Example:

```
#include <stdio.h>
#include <time.h>

void main()
{
    clock_t  cstart, cend;
    cstart = clock();
    /*  .
        .
        .
    */
    cend = clock();
    printf( "%4.2f seconds to calculate\n",
            ((float)cend - cstart) / CLOCKS_PER_SEC );
}
```

The following 32-bit code is generated by the Watcom C compiler (wcc386) using the "fpc" option.

```
main_   push    ebx
        push    edx
        call    clock_
        mov     edx,eax
        call    clock_
        call    __U4FS  ; unsigned 4 to floating single
        mov     ebx,eax
        mov     eax,edx
        call    __U4FS  ; unsigned 4 to floating single
        mov     edx,eax
        mov     eax,ebx
        call    __FSS   ; floating single subtract
        mov     edx,3c23d70aH
        call    __FSM   ; floating single multiply
        call    __FSFD  ; floating single to floating double
        push    edx
        push    eax
        push    offset L1
        call    printf_
        add     esp,0000000cH
        pop     edx
        pop     ebx
        ret
```

The following 32-bit code is generated by the Watcom C compiler (wcc386) using the "fpi" option.

```
main_   push    ebx
        push    edx
        sub     esp,00000010H
        call    clock_
        mov     edx,eax
        call    clock_
        xor     ebx,ebx
        mov     [esp],eax
        mov     +4H[esp],ebx
        mov     +8H[esp],edx
        mov     +0cH[esp],ebx
        fild    qword ptr [esp]     ; integer to double
        fild    qword ptr +8H[esp]  ; integer to double
        fsubp   st(1),st            ; subtract
        fmul    dword ptr L2        ; multiply
        sub     esp,00000008H
        fstp    qword ptr [esp]     ; store into memory
        push    offset L1
        call    printf_
        add     esp,0000000cH
        add     esp,00000010H
        pop     edx
        pop     ebx
        ret
```

fpc

All floating-point arithmetic is done with calls to a floating-point emulation library. If a numeric data processor is present in the system, it will be used by the library; otherwise floating-point operations are simulated in software. This option should be used for any of the following reasons:

1. Speed of floating-point emulation is favoured over code size.
2. An application containing floating-point operations is to be stored in ROM and an 80x87 will not be present in the system.

The macro `__SW_FPC` will be predefined if "fpc" is selected.

> ***Note:*** When any module in an application is compiled with the "fpc" option, then all modules must be compiled with the "fpc" option.
>
> Different math libraries are provided for applications which have been compiled with a particular floating-point option. See the section entitled "Watcom C/C++ Math Libraries" on page 126.

See the section entitled "The NO87 Environment Variable" on page 129 for information on testing the floating-point simulation code on personal computers equipped with a coprocessor.

fpi

(16-bit only) The compiler will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. Depending on which library the code is linked against, these instructions will be left as is or they will be replaced by special interrupt instructions. In the latter case, floating-point will be emulated if an 80x87 is not present. This is the default floating-point option if none is specified.

(32-bit only) The compiler will generate in-line 387-compatible numeric data processor instructions into the object code for floating-point operations. When any module containing floating-point operations is compiled with the "fpi" option, coprocessor emulation software will be included in the application when it is linked.

For 32-bit Watcom Windows-extender applications or 32-bit applications run in Windows 3.1 DOS boxes, you must also include the `WEMU387.386` file in the `[386enh]` section of the `SYSTEM.INI` file.

Example:

```
device=C:\WATCOM\binw\wemu387.386
```

Note that the `WDEBUG.386` file which is installed by the Watcom Installation software contains the emulation support found in the `WEMU387.386` file.

Thus, a math coprocessor need not be present at run-time. This is the default floating-point option if none is specified. The macros `__FPI__` and `__SW_FPI` will be predefined if "fpi" is selected.

> ***Note:*** When any module in an application is compiled with a particular "floating-point" option, then all modules must be compiled with the same option.
>
> If you wish to have floating-point emulation software included in the application, you should select the "fpi" option. A math coprocessor need not be present at run-time.
>
> Different math libraries are provided for applications which have been compiled with a particular floating-point option. See the section entitled "Watcom C/C++ Math Libraries" on page 126.

See the section entitled "The NO87 Environment Variable" on page 129 for information on testing the math coprocessor emulation code on personal computers equipped with a coprocessor.

fpi87

(16-bit only) The compiler will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. An 8087 or compatible math coprocessor must be present at run-time. If the "2" option is used in conjunction with this option, the compiler will generate 287 and upwards compatible instructions; otherwise, the compiler will generate 8087 compatible instructions.

(32-bit only) The compiler will generate in-line 387-compatible numeric data processor instructions into the object code for floating-point operations. When the "fpi87" option is used exclusively, coprocessor emulation software is not included in the application when it is linked. A 387 or compatible math coprocessor must be present at run-time.

The macros `__FPI__` and `__SW_FPI87` will be predefined if "fpi87" is selected. See Note with description of "fpi" option.

fp2

The compiler will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. For Watcom compilers generating 16-bit code, this option is the default. For 32-bit applications, use this option if you wish to support those few 386 systems that are equipped with a 287 numeric data processor ("fp3" is the default for Watcom compilers generating 32-bit code). However, for 32-bit applications, the use of this option will reduce execution performance. Use this option in conjunction with the "fpi" or "fpi87" options. The macro `__SW_FP2` will be predefined if "fp2" is selected.

fp3

The compiler will generate in-line 387-compatible numeric data processor instructions into the object code for floating-point operations. For 16-bit applications, the use of this option will limit the range of systems on which the application will run but there are execution performance improvements. For Watcom compilers generating 32-bit code, this option is the default. Use this option in conjunction with the "fpi" or "fpi87" options. The macro `__SW_FP3` will be predefined if "fp3" is selected.

fp5

The compiler will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. The sequence of floating-point instructions will be optimized for greatest possible performance on the Intel Pentium processor. For 16-bit applications, the use of this option will limit the range of systems on which the application will run but there are execution performance improvements. Use this option in conjunction with the "fpi" or "fpi87" options. The macro `__SW_FP5` will be predefined if "fp5" is selected.

fp6

The compiler will generate in-line 80x87 numeric data processor instructions into the object code for floating-point operations. The sequence of floating-point instructions will be optimized for greatest possible performance on the Intel Pentium Pro processor. For 16-bit applications, the use of this option will limit the range of systems on which the application will run but there are execution performance improvements. Use this option in conjunction with the "fpi" or "fpi87" options. The macro `__SW_FP6` will be predefined if "fp6" is selected.

fpd

A subtle problem was detected in the FDIV instruction of Intel's original Pentium CPU. In certain rare cases, the result of a floating-point divide could have less precision than it should. Contact Intel directly for more information on the issue.

As a result, the run-time system startup code has been modified to test for a faulty Pentium. If the FDIV instruction is found to be flawed, the low order bit of the run-time system variable `__chipbug` will be set.

```
extern unsigned __near __chipbug;
```

If the FDIV instruction does not show the problem, the low order bit will be clear. If the Pentium FDIV flaw is a concern for your application, there are two approaches that you could take:

1. You may test the `__chipbug` variable in your code in all floating-point and memory models and take appropriate action (such as display a warning message or discontinue the application).

2. Alternately, you can use the "fpd" option when compiling your code. This option directs the compiler to generate additional code whenever an FDIV instruction is generated which tests the low order bit of `__chipbug` and, if on, calls the software workaround code in the math libraries. If the bit is off, an in-line FDIV instruction will be performed as before.

If you know that your application will never run on a defective Pentium CPU, or your analysis shows that the FDIV problem will not affect your results, you need not use the "fpd" option. The macro `__SW_FPD` will be predefined if "fpd" is selected.

2.3.8 Segments/Modules

This group of options deals with object file data structures that are generated by the compiler.

g=<codegroup>

The generated code is placed in the group called "<codegroup>". The default "text" segment name will be "<codegroup>_TEXT" but this can be overridden by the "nt" option.

Example:

```
C>compiler_name report /g=RPTGROUP /s
```

(16-bit only) <<

This is useful when compiling applications for small code models where the total application will contain more than 64 kilobytes of code. Each group can contain up to 64 kilobytes of code. The application follows a "mixed" code model since it contains a mix of small and large code (intra-segment and inter-segment calls). Memory models are described in the chapter entitled "16-bit Memory Models" on page 135. The `far` keyword is used to describe

routines that are referenced from one group/segment but are defined in another group/segment.

For small code models, the "s" option should be used in conjunction with the "g" option to prevent the generation of calls to the C run-time "stack overflow check" routine (__STK). You must also avoid calls to other "small code" C run-time library routines since inter-segment "near" calls to C library routines are not possible.

>> (16-bit only)

nc=<name>

The default "code" class name is "CODE". The small code model "_TEXT" segment and the large code model "module_name_TEXT" segments belong to the "CODE" class. This option allows you to select a different class name for these code segments. The name of the "code" class is explicitly set to "<name>".

Note that the default "data" class names are "DATA" (for the "CONST", "CONST2" and "_DATA" segments) and "BSS" (for the "_BSS" segment). There is no provision for changing the data class names.

nd=<name>

This option permits you to define a special prefix for the "CONST", "CONST2", "_DATA", and "_BSS" segment names. The name of the group to which these segments belong is also changed from "DGROUP" to "<name>_GROUP". This option is especially useful in the creation of 16-bit Dynamic Link Library (DLL) routines.

Example:

```
C>compiler_name report /nd=spec
```

In the above example, the segment names become "specCONST", "specCONST2", "spec_DATA", and "spec_BSS" and the group name becomes "spec_GROUP".

By default, the data group "DGROUP" consists of the "CONST", "CONST2", "_DATA", and "_BSS" segments. The compiler places certain types of data in each segment. The "CONST" segment contains constant literals that appear in your source code.

Example:

```
char *birds[ 3 ] = { "robin", "finch", "wren" };

printf( "Hello world\n" );
```

In the above example, the strings "Hello world\n", "robin", "finch", etc. appear in the "CONST" segment.

The "CONST2" segment contains initialized read-only data.

Example:

```
extern const int cvar = 1;
int var = 2;
int table[ 5 ] = { 1, 2, 3, 4, 5 };
char *birds[ 3 ] = { "robin", "finch", "wren" };
```

In the above example, the constant variable `cvar` is placed in the "CONST2" segment by the 16-bit C compiler and the 16-bit and 32-bit C++ compilers.

The "_BSS" segment contains uninitialized data such as scalars, structures, or arrays.

Example:

```
int var1;
int array1[ 400 ];
```

Other data segments containing data, specifically declared to be `far` or exceeding the data threshold (see "zt" option), are named either "module_nameN_DATA" when using the C compiler or "module_name_DATAN" when using the C++ compiler where "N" is some integral number.

Example:

```
int far array2[400];
```

In the above example, `array2` is placed in the segment "report11_DATA" (C) or "report_DATA11" (C++) provided that the module name is "report".

The macro `__SW_ND` will be predefined if "nd" is selected.

nm=<name>

By default, the object file name and the module name that is placed within it are constructed from the source file name. When the "nm" option is used, the module name that is placed into the object file is "<name>". For large code models, the "text" segment name is "<name>_TEXT" unless the "nt" option is used.

In the following example, the preprocessed output from `REPORT.C` is stored on drive "D" under the name `TEMP.C`. The file is compiled with the "nm" option so that the module name imbedded into the object file is "REPORT" rather than "TEMP".

Example:

```
C>compiler_name report /pl /fo=d:\temp.c
C>compiler_name d:\temp /nm=report /fo=report
```

Since the "fo" option is also used, the resultant object file is called `REPORT.OBJ`.

nt=<name>

The name of the "text" segment is explicitly set to "<name>". By default, the "text" segment name is "_TEXT" for small code models and "module_name_TEXT" for large code models.

Application Type	Memory Model	Code Segment
16-bit	tiny	_TEXT
32-bit	flat	_TEXT
16/32-bit	small	_TEXT
16/32-bit	medium	module_name_TEXT
16/32-bit	compact	_TEXT
16/32-bit	large	module_name_TEXT
16-bit	huge	module_name_TEXT

zm

The "zm" option instructs the code generator to place each function into a separate segment.

In small code models, the segment name is "_TEXT" by default.

(C only) In large code models, the segment name is composed of the function name concatenated with the string "_TEXT".

(C++ only) In large code models, the segment name is composed of the module name concatenated with the string "_TEXT" and a unique integral number.

The default string "_TEXT" can be altered using the "nt" option (see "nt=<name>").

The advantages to this option are:

1. Since each function is placed in its own segment, functions that are not required by an application are omitted from the executable by the linker (when "OPTION ELIMINATE" is specified).
2. This can result in smaller executables.
3. This benefit applies to both small and large code models.
4. This option allows you to create granular libraries without resorting to placing each function in a separate file.

Example:

```
static int foo( int x )
{
    return x - 1;
}

static int near foo1( int x )
{
    return x + 1;
}

int foo2( int y )
{
    return foo(y) * foo1(y-1);
}

int foo3( int x, int y )
{
    return x + y * x;
}
```

The disadvantages to this option are:

1. The "near call" optimization for static functions in large code models is disabled (e.g., the function `foo` in the example above will never be "near called". Static functions will always be "far called" in large code models.
2. Near static functions will still be "near called" (e.g., the function `foo1` is "near called" in the example above). However, this can lead to problems at link time if the caller function ends up in a different segment from the called function (the linker will issue a message if this is the case).
3. The "common epilogue" optimization is lost.
4. The linker "OPTION ELIMINATE" must be specified when linking an application to take advantage of the granularity inherent in object files/libraries compiled with this option.

This option can be used in paging environments where special segment ordering may be employed. The "alloc_text" pragma is often used in conjunction with this option to place functions into specific segments.

The macro `__SW_ZM` will be predefined if "zm" is selected.

zmf

(C++ only) This option is identical to the "zm" option (see "zm" on page 58) except for the following large code model consideration.

Instead of placing each function in a segment with a different name, the code generator will place each function in a segment with the same name (the name of the module suffixed by "_TEXT").

The advantages to this option are:

1. All functions in a module will reside in the same physical segment in an executable.
2. The "near call" optimization for static functions in large code models is not disabled (e.g., the function `foo` in the example above will be "near called". Static functions will always be "near called" in large code models.
3. The problem associated with calling "near" functions goes away since all functions in a module will reside in the same physical segment (e.g., the function `foo1` is "near" in the example above).

The disadvantages to this option are:

1. The size of a physical segment is restricted to 64K in 16-bit applications. Although this may limit the number of functions that can be placed in the segment, the restriction is only on a "per module" basis.
2. Although less constricting, the size of a physical segment is restricted to 4G in a 32-bit application.

2.3.9 80x86 Run-time Conventions

This group of options deals with the 80x86 run-time environment.

0

(16-bit only) The compiler will make use of only 8086 instructions in the generated object code. This is the default. The resulting code will run on 8086 and all upward compatible processors. The macro `__SW_0` will be predefined if "0" is selected.

1

(16-bit only) The compiler will make use of 186 instructions in the generated object code whenever possible. The resulting code probably will not run on 8086 compatible processors but it will run on 186 and upward compatible processors. The macro `__SW_1` will be predefined if "1" is selected.

2

(16-bit only) The compiler will make use of 286 instructions in the generated object code whenever possible. The resulting code probably will not run on 8086 or 186 compatible

processors but it will run on 286 and upward compatible processors. The macro `__SW_2` will be predefined if "2" is selected.

3

(16-bit only) The compiler will make use of some 386 instructions and FS or GS (if "zff" or "zgf" options are used) in the generated object code whenever possible. The code will be optimized for 386 processors. The resulting code probably will not run on 8086, 186 or 286 compatible processors but it will run on 386 and upward compatible processors. The macro `__SW_3` will be predefined if "3" is selected.

4

(16-bit only) The compiler will make use of some 386 instructions and FS or GS (if "zff" or "zgf" options are used) in the generated object code whenever possible. The code will be optimized for 486 processors. The resulting code probably will not run on 8086, 186 or 286 compatible processors but it will run on 386 and upward compatible processors. The macro `__SW_4` will be predefined if "4" is selected.

5

(16-bit only) The compiler will make use of some 386 instructions and FS or GS (if "zff" or "zgf" options are used) in the generated object code whenever possible. The code will be optimized for the Intel Pentium processor. The resulting code probably will not run on 8086, 186 or 286 compatible processors but it will run on 386 and upward compatible processors. The macro `__SW_5` will be predefined if "5" is selected.

6

(16-bit only) The compiler will make use of some 386 instructions and FS or GS (if "zff" or "zgf" options are used) in the generated object code whenever possible. The code will be optimized for the Intel Pentium Pro processor. The resulting code probably will not run on 8086, 186 or 286 compatible processors but it will run on 386 and upward compatible processors. The macro `__SW_6` will be predefined if "6" is selected.

3{r|s}

(32-bit only) The compiler will generate 386 instructions based on 386 instruction timings (see "4", "5" and "6" below).

If the "r" suffix is specified, the following machine-level code strategy is employed.

- The compiler will pass arguments in registers whenever possible. This is the default method used to pass arguments (unless the "bt=netware" option is specified).
- All registers except EAX are preserved across function calls.
- When any form of the "fpi" option is used, the result of functions of type "float" and "double" is returned in ST(0).
- When the "fpc" option is used, the result of a function of type "float" is returned in EAX and the result of a function of type "double" is returned in EDX:EAX.
- The resulting code will be smaller than that which is generated for the stack-based method of passing arguments (see "3s" below).
- The default naming convention for all global functions is such that an underscore character ("_") is suffixed to the symbol name. The default naming convention for all global variables is such that an underscore character ("_") is prefixed to the symbol name.

If the "s" suffix is specified, the following machine-level code strategy is employed.

- The compiler will pass all arguments on the stack.
- The EAX, ECX and EDX registers are not preserved across function calls.
- The FS and GS registers are not preserved across function calls.
- The result of a function of type "float" is returned in EAX. The result of a function of type "double" is returned in EDX:EAX.
- The resulting code will be larger than that which is generated for the register method of passing arguments (see "3r" above).
- The naming convention for all global functions and variables is modified such that no underscore characters ("_") are prefixed or suffixed to the symbol name.

The "s" conventions are similar to those used by the MetaWare High C 386 compiler.

By default, "r" is selected if neither "r" nor "s" is specified.

The macro `__SW_3` will be predefined if "3" is selected. The macro `__SW_3R` will be predefined if "r" is selected (or defaulted). The macro `__SW_3S` will be predefined if "s" is selected.

4{r|s}

(32-bit only) This option is identical to "3{r|s}" except that the compiler will generate 386 instructions based on 486 instruction timings. The code is optimized for 486 processors rather than 386 processors. By default, "r" is selected if neither "r" nor "s" is specified. The macro `__SW_4` will be predefined if "4" is selected. The macro `__SW_3R` will be predefined if "r" is selected (or defaulted). The macro `__SW_3S` will be predefined if "s" is selected.

5{r|s}

(32-bit only) This option is identical to "3{r|s}" except that the compiler will generate 386 instructions based on Intel Pentium instruction timings. This is the default. The code is optimized for Intel Pentium processors rather than 386 processors. By default, "r" is selected if neither "r" nor "s" is specified. The macro `__SW_5` will be predefined if "5" is selected. The macro `__SW_3R` will be predefined if "r" is selected (or defaulted). The macro `__SW_3S` will be predefined if "s" is selected.

6{r|s}

(32-bit only) This option is identical to "3{r|s}" except that the compiler will generate 386 instructions based on Intel Pentium Pro instruction timings. The code is optimized for Intel Pentium Pro processors rather than 386 processors. By default, "r" is selected if neither "r" nor "s" is specified. The macro `__SW_6` will be predefined if "6" is selected. The macro `__SW_3R` will be predefined if "r" is selected (or defaulted). The macro `__SW_3S` will be predefined if "s" is selected.

mf

(32-bit only) The "flat" memory model (code and data up to 4 gigabytes) is selected. By default, the 32-bit compiler will select this memory model unless the target system is Netware in which case "small" is selected. The following macros will be predefined.

```
M_386FM
_M_386FM
__FLAT__
```

ms

The "small" memory model (small code, small data) is selected. By default, the 16-bit compiler will select this memory model. When the target system is Netware, the 32-bit compiler will select this memory model. The following macros will be predefined.

```
M_I86SM
_M_I86SM
M_386SM
_M_386SM
__SMALL__
```

mm

The "medium" memory model (big code, small data) is selected. The following macros will be predefined.

```
M_I86MM
__MEDIUM__
```

mc

The "compact" memory model (small code, big data) is selected. The following macros will be predefined.

```
M_I86CM
_M_I86CM
__COMPACT__
```

ml

The "large" memory model (big code, big data) is selected. The following macros will be predefined.

```
M_I86LM
__LARGE__
```

mh

(16-bit only) The "huge" memory model (big code, huge data) is selected. The following macros will be predefined.

```
M_I86HM
__HUGE__
```

Memory models are described in the chapters entitled "16-bit Memory Models" on page 135 and "32-bit Memory Models" on page 217. Other architectural aspects of the Intel 86 family such as pointer size are discussed in the sections entitled "Sizes of Predefined Types" on page 148 in the chapter entitled "16-bit Assembly Language Considerations" or "Sizes of Predefined Types" on page 230 in the chapter entitled "32-bit Assembly Language Considerations"

zd{f,p}

The "zdf" option allows the code generator to use the DS register to point to other segments besides "DGROUP" This is the default in the 16-bit compact, large, and huge memory models (except for 16-bit Windows applications).

The "zdp" option informs the code generator that the DS register must always point to "DGROUP" This is the default in the 16-bit small and medium memory models, all of the 16-bit Windows memory models, and the 32-bit small and flat memory models. The macro `__SW_ZDF` will be predefined if "zdf" is selected. The macro `__SW_ZDP` will be predefined if "zdp" is selected.

zdl

(32-bit only) The "zdl" option causes generation of code to load the DS register directly from DGROUP (rather than the default run-time call). This option causes the generation of a segment relocation. This option is used with the "zdp" option but not the "zdf" option.

zf{f,p}

The "zff" option allows the code generator to use the FS register (default for all but flat memory model). The "zfp" option informs the code generator that the FS register must not be used (default in flat memory model). The macro `__SW_ZFF` will be predefined if "zff" is selected. The macro `__SW_ZFP` will be predefined if "zfp" is selected.

zg{f,p}

The "zgf" option allows the code generator to use the GS register (default for all memory models). The "zgp" option informs the code generator that the GS register must not be used. The macro `__SW_ZGF` will be predefined if "zgf" is selected. The macro `__SW_ZGP` will be predefined if "zgp" is selected.

zu

The "zu" option relaxes the restriction that the SS register contains the base address of the default data segment, "DGROUP". Normally, all data items are placed into the group "DGROUP" and the SS register contains the base address of this group. When the "zu" option is selected, the SS register is volatile (assumed to point to another segment) and any global data references require loading a segment register such as DS with the base address of "DGROUP".

(16-bit only) This option is useful when compiling routines that are to be placed in a Dynamic Link Library (DLL) since the SS register points to the stack segment of the calling application upon entry to the function.

The macro `__SW_ZU` will be predefined if "zu" is selected.

2.3.10 Optimizations

When specified on the command line, optimization options may be specified individually (oa, oi) or the letters may be strung together (oailt).

oa

Alias checking is relaxed. When this option is specified, the code optimizer will assume that global variables are not indirectly referenced through pointers. This assumption may reduce the size of the code that is generated. The following example helps to illustrate this point.

Example:

```
extern int i;

void rtn( int *pi )
{
    int k;
    for( k = 0; k < 10; ++k ) {
        (*pi)++;
        i++;
    }
}
```

In the above example, if "i" and "*pi" referenced the same integer object then "i" would be incremented by 2 each time through the "for" loop and we would call the pointer reference "*pi" an alias for the variable "i". In this situation, the compiler could not bind the variable "i" to a register without making sure that the "in-memory" copy of "i" was kept up-to-date. In most cases, the above situation does not arise. Rarely would we reference the same variable directly by name and indirectly through a pointer in the same routine. The "oa" option instructs the code generator that such cases do not arise in the module to be compiled. The code generator will be able to produce more efficient code when it does not have to worry about the alias "problem".

The macro `__SW_OA` will be predefined if "oa" is selected.

ob

When the "ob" option is specified, the code generator will try to order the blocks of code emitted such that the "expected" execution path (as determined by a set of simple heuristics) will be straight through, with other cases being handled by jumps to separate blocks of code "out of line". This will result in better cache utilization on the Pentium. If the heuristics do not apply to your code, it could result in a performance decrease.

oc

This option may be used to disable the optimization where a "CALL" followed by a "RET" (return) is changed into a "JMP" (jump) instruction.

(16-bit only) This option is required if you wish to link an overlayed program using the Microsoft DOS Overlay Linker. The Microsoft DOS Overlay Linker will create overlay calls for a "CALL" instruction only. This option is not required when using the Watcom Linker.

The macro `__SW_OC` will be predefined if "oc" is selected.

od

Non-optimized code sequences are generated. The resulting code will be much easier to debug when using the Watcom Debugger. By default, the compiler will select "od" if "d2" is specified. If "d2" is followed by one of the other "o?" options then "od" is overridden.

Example:

```
C>compiler_name report /d2 /os
```

The macro `__SW_OD` will be predefined if "od" is selected.

oe=<num>

Certain user functions are expanded in-line. The criteria for which functions are selected for in-line expansion is based on the "size" of the function in terms of the number of "quads" generated by the function. A quad (quadruple) consists of four components: an operator, two arguments and a result (hence the name). All expressions can be decomposed into a series of quads. For example, the statement `a = -b * (c + d)` can be broken down into the following quads.

```
OP      ARG1    ARG2    RESULT
------- ------- ------- ------
uminus  b       _       t1
+       c       d       t2
*       t1      t2      t3
=       t3      _       a
```

The number of "quads" generated corresponds closely with the number of operators used in an expression. Functions which require more than "<num>" quads are not expanded in-line. The default number is 20. This optimization is useful when locally-referenced functions are small in size.

Example:

```
C>compiler_name dhrystone /oe
```

oh

This option enables repeated optimizations (which can result in longer compiles).

oi

Certain library functions are generated in-line. You must include the appropriate header file containing the prototype for the desired function so that it will be generated in-line. The functions that can be generated in-line are:

```
abs         _disable    div         _enable     fabs
_fmemchr    _fmemcmp    _fmemcpy    _fmemset    _fstrcat
_fstrcmp    _fstrcpy    _fstrlen    inpd (2)    inpw
inp         labs        ldiv (2)    _lrotl (2)  _lrotr (2)
memchr      memcmp      memcpy      memset (1)  movedata
outpd (2)   outpw       outp        _rotl       _rotr
strcat      strchr      strcmp (1)  strcpy      strlen
```

*1 16-bit only
*2 32-bit only

The macros `__INLINE_FUNCTIONS__` and `__SW_OI` will be predefined if "oi" is selected.

oi+

(C++ only) This option encompasses "oi" but also sets *inline_depth* to its maximum (255). By default, *inline_depth* is 3. The *inline_depth* can also be changed by using the C++ `inline_depth` pragma.

ok

This option enables flowing of register save (from prologue) down into the function's flow graph. This means that register save/restores will not be executed when it is not necessary (as can be the case when a function consists of an if-else construct with a simple part that does little and a more complex part that does a lot).

ol

Loop optimizations are performed. This includes moving loop-invariant expressions outside the loops. The macro `__SW_OL` will be predefined if "ol" is selected.

ol+

Loop optimizations are performed including loop unrolling. This includes moving loop-invariant expressions outside the loops and turning some loops into straight-line code. The macro `__SW_OL` will be predefined if "ol+" is selected.

om

Generate in-line 80x87 code for math functions like sin, cos, tan, etc. If this option is selected, it is the programmer's responsibility to make sure that arguments to these functions are within the range accepted by the `fsin, fcos,` etc. instructions since no run-time check is made. For 16-bit, you must also include the "fp3" option to get in-line 80x87 code (except for fabs). The functions that can be generated in-line are:

```
atan  cos   exp   fabs  log10 log   sin   sqrt  tan
```

The macro `__SW_OM` will be predefined if "om" is selected.

on

This option allows the compiler to replace floating-point divisions with multiplications by the reciprocal. This generates faster code, but the result may not be the same because the reciprocal may not be exactly representable. The macro `__SW_ON` will be predefined if "on" is selected.

oo

By default, the compiler will abort compilation if it runs low on memory. This option forces the compiler to continue compilation even when low on memory, however, this can result in very poor code being generated. The macro `__SW_OO` will be predefined if "oo" is selected.

op

This option causes the compiler to store intermediate floating-point results into memory in order to generate consistent floating-point results rather than keeping values in the 80x87 registers where they have more precision. The macro `__SW_OP` will be predefined if "op" is selected.

or

This option enables reordering of instructions (instruction scheduling) to achieve better performance on pipelined architectures such as the Intel 486 and Pentium processors. This option is essential for generating fast code for the Intel Pentium processor. Selecting this option will make it slightly more difficult to debug because the assembly language instructions generated for a source statement may be intermixed with instructions generated for surrounding statements. The macro `__SW_OR` will be predefined if "or" is selected.

os

Space is favoured over time when generating code (smaller code but possibly slower execution). By default, the compiler selects a balance between "space" and "time". The macro `__SW_OS` will be predefined if "os" is selected.

ot

Time is favoured over space when generating code (faster execution but possibly larger code). By default, the compiler selects a balance between "space" and "time". The macro `__SW_OT` will be predefined if "ot" is selected.

ou

This option forces the compiler to make sure that all function labels are unique. Thus the compiler will not place two function labels at the same address even if the code for the two functions are identical. This option is automatically selected if the "za" option is specified. The macro `__SW_OU` will be predefined if "ou" is selected.

ox

The "obiklmr" and "s" (no stack overflow checking) options are selected.

oz

This option prevents the compiler from omitting NULL pointer checks on pointer conversions. By default, the compiler omits NULL pointer checks on pointer conversions when it is safe to do so. Consider the following example.

```
struct B1 {
    int b1;
};
struct B2 {
    int b2;
};
struct D : B1, B2 {
    int d;
};

void clear_D( D *p )
{
        p->d = 0;
        B1 *p1 = p;
        p1->b1 = 0;
        B2 *p2 = p;
        p2->b2 = 0;
}
```

In this example, the C++ compiler must ensure that `p1` and `p2` become NULL if `p` is NULL (since no offset adjustment is allowed for a NULL pointer). However, the first executable statement implies that `p` is not NULL since, in most operating environments, the executing program would crash at the first executable statement if `p` was NULL. The "oz" option will prevent the compiler from omitting the check for a NULL pointer.

The macro __SW_OZ will be predefined if "oz" is selected.

When "ox" is combined with the "on", "oe", "oa" and "ot" options ("oneatx") and the "zp4" option, the code generator will attempt to give you the fastest executing code possible irrespective of architecture. Other options can give you architecture specific optimizations to further improve the speed of your code. Note that specifying "oneatx" is equivalent to specifying "oneatilmr" and "s". See the section entitled "Benchmarking Hints" on page 81 for more information on generating fast code.

2.3.11 C++ Exception Handling

The "xd..." options disable exception handling. Consequently, it is not possible to use ***throw, try,*** or ***catch*** statements, or to specify a function exception specification. If your program (or a library which it includes) throws exceptions, then one of the "xs..." options should be used to compile all the modules in your program; otherwise, any active objects created within the module will not be destructed during exception processing.

Multiple schemes are possible, allowing experimentation to determine the optimal scheme for particular circumstances. You can mix and match schemes on a module basis, with the proviso that exceptions should be enabled wherever it is possible that a created object should be destructed by the exception mechanism.

xd

This option disables exception handling. It is the default option if no exception handling option is specified. When this option is specified (or defaulted):

- Destruction of objects is caused by direct calls to the appropriate destructors
- Destructor functions are implemented with direct calls to appropriate destructors to destruct base classes and class members.

xdt

This option is the same as "xd" (see "xd").

xds

This option disables exception handling. When this option is specified:

- Destruction of objects is caused by direct calls to the appropriate destructors.
- Destruction of base classes and class members is accomplished by interpreting tables.

- This option, in general, generates smaller code, with increased execution time and with more run-time system routines included by the linker.

xs

This option enables exception handling using a balanced scheme. When this option is specified:

- Tables are interpreted to effect destruction of temporaries and automatic objects; destructor functions are implemented with direct calls to appropriate destructors to destruct base classes and class members.

xst

This option enables exception handling using a time-saving scheme. When this option is specified:

- Destruction of temporaries and automatic objects is accomplished with direct calls to appropriate destructors; destructor functions are implemented with direct calls to appropriate destructors to destruct base classes and class members.
- This scheme will execute faster, but will use more space in general.

xss

This option enables exception handling using a space-saving scheme. When this option is specified:

- Tables are interpreted to effect destruction of temporaries and automatic objects; destruction of base classes and class members is accomplished by interpreting tables.
- This option, in general, generates smaller code, with increased execution time.

2.3.12 Double-Byte/Unicode Characters

This group of options deals with compile-time aspects of character sets used in the source code.

zk{0,1,2,l}

This option causes the compiler to recognize double-byte characters in strings. When the compiler scans a text string enclosed in quotes ("), it will recognize the first byte of a double-byte character and suppress lexical analysis of the second byte. This will prevent the compiler from misinterpreting the second byte as a "\" or quote (") character.

zk, zk0 These options cause the compiler to process strings for Japanese double-byte characters (range 0x81 - 0x9F and 0xE0 - 0xFC). The characters in the range A0 - DF are single-byte Katakana.

zk1 This option causes the compiler to process strings for Traditional Chinese and Taiwanese double-byte characters (range 0x81 - 0xFC).

zk2 This option causes the compiler to process strings for Korean Hangeul double-byte characters (range 0x81 - 0xFD).

zkl This option causes the compiler to process strings using the current code page. If the local character set includes double-byte characters then string processing will check for lead bytes.

The macro `__SW_ZK` will be predefined if any "zk" option is selected.

zk0u

This option causes the compiler to process strings for Japanese double-byte characters (range 0x81 - 0x9F and 0xE0 - 0xFC). The characters in the range A0 - DF are single-byte Katakana. All characters, including Kanji, in wide characters (L'c') and wide strings (L"string") are translated to UNICODE.

When the compiler scans a text string enclosed in quotes ("), it will recognize the first byte of a double-byte character and suppress lexical analysis of the second byte. This will prevent the compiler from misinterpreting the second byte as a "\" or quote (") character.

zku=<codepage>

Characters in wide characters (L'c') and wide strings (L"string") are translated to UNICODE. The UNICODE translate table for the specified code page is loaded from a file with the name "UNICODE.cpn" where "cpn" is the code page number (e.g., zku=850 selects file "UNICODE.850"). The compiler locates this file by searching the paths listed in the **PATH** environment variable.

2.3.13 Compatibility with Microsoft Visual C++

This group of options deals with compatibility with Microsoft's Visual C++ compiler.

vc...

The "vc" option prefix is used to introduce a set of Microsoft Visual C++ compatibility options. At present, there is only one: vcap.

vcap

This options tells the compiler to allow ***_alloca()*** to be used in a parameter list. The optimizer has to do extra work to allow this but since it is rare (and easily worked around if you can), you have to ask the optimizer to handle this case. You also may get less efficient code in some cases.

2.3.14 Compatibility with Older Versions of the 80x86 Compilers

This group of options deals with compatibility with older versions of Watcom's 80x86 compilers.

r

This option instructs the compiler to generate function prologue and epilogue sequences that save and restore any segment registers that are modified by the function. Caution should be exercised when using this option. If the value of the segment register being restored matches the value of a segment that was freed within the function, a general protection fault will occur in protected-mode environments. By default, the compiler does not generate code to save and restore segment registers. This option is provided for compatibility with the version 8.0 release. The macro `__SW_R` will be predefined if "r" is selected.

fpr

Use this option if you want to generate floating-point instructions that will be compatible with version 9.0 or earlier of the compilers. For more information on floating-point conventions see the sections entitled "Using the 80x87 to Pass Arguments" on page 209 and "Using the 80x87 to Pass Arguments" on page 296.

zz

Use this option if you want to generate __stdcall function names that will be compatible with version 10.0 of the compilers. When this option is omitted, all C symbols (extern "C" symbols in C++) are suffixed by "@nnn" where "nnn" is the sum of the argument sizes (each size is rounded up to a multiple of 4 bytes so that char and short are size 4). When the argument list contains "...", the "@nnn" suffix is omitted. This convention is compatible with Microsoft. For more information on the __stdcall convention see the section entitled "Watcom C/C++ Extended Keywords" on page 95.

3 The Watcom C/C++ Compilers

This chapter covers the following topics.

- Command line syntax (see "Watcom C/C++ Command Line Format")
- Environment variables used by the compilers (see "Environment Variables" on page 77)
- Examples of command line syntax (see "Watcom C/C++ Command Line Examples" on page 78)
- Interpreting diagnostic messages (see "Compiler Diagnostics" on page 83)
- #include file handling (see "Watcom C/C++ #include File Processing" on page 85)
- Using the preprocessor built into the compilers (see "Watcom C/C++ Preprocessor" on page 88)
- System-dependent macros predefined by the compilers (see "Watcom C/C++ Predefined Macros" on page 89)
- Additional keywords supported by the compilers (see "Watcom C/C++ Extended Keywords" on page 95)
- Based pointer support in the compilers (see "Based Pointers" on page 104)
- Notes about the Code Generator (see "The Watcom Code Generator" on page 115)

3.1 Watcom C/C++ Command Line Format

The formal Watcom C/C++ command line syntax is shown below.

```
compiler_name [options] [file_spec] [options] [@extra_opts]
```

The square brackets [ ] denote items which are optional.

compiler_name is one of the Watcom C/C++ compiler command names.

WCC	is the Watcom C compiler for 16-bit Intel platforms.
WPP	is the Watcom C++ compiler for 16-bit Intel platforms.
WCC386	is the Watcom C compiler for 32-bit Intel platforms.
WPP386	is the Watcom C++ compiler for 32-bit Intel platforms.

file_spec is the file name specification of one or more files to be compiled.

If no drive is specified, the default drive is assumed.

If no path is specified, the current working directory is assumed. If the file is not in the current directory, an adjacent "C" directory (i.e., `..\C`) is searched if it exists.

If no file extension is specified, the compiler will check for a file with one of the following extensions in the order listed:

.CPP	(C++ only)
.CC	(C++ only)
.C	(C/C++)

If a period "." is specified but not the extension, the file is assumed to have no filename extension.

If only the compiler name is specified then the compiler will display a list of available options.

options is a list of valid compiler options, each preceded by a slash ("/") or a dash ("–"). Options may be specified in any order.

extra_opts is the name of an environment variable or file which contains additional command line options to be processed. If the specified environment variable does not exist, a search is made for a file with the specified name. If no file extension is included in the specified name, the default file extension is ".occ". A search of the current directory is made. If not successful, an adjacent "OCC" directory (i.e., `..\OCC`) is searched if it exists.

3.2 Watcom C/C++ DLL-based Compilers

The compilers are also available in Dynamic Link Library (DLL) form.

WCCD is the DLL version of the Watcom C compiler for 16-bit Intel platforms.
WPPDI86 is the DLL version of the Watcom C++ compiler for 16-bit Intel platforms.
WCCD386 is the DLL version of the Watcom C compiler for 32-bit Intel platforms.
WPPD386 is the DLL version of the Watcom C++ compiler for 32-bit Intel platforms.

The DLL versions of the compilers can be loaded from the Watcom Integrated Development Environment (IDE) and Watcom Make.

3.3 Environment Variables

Environment variables can be used to specify commonly used compiler options. There is one environment variable for each compiler (the name of the environment variable is the same as the compiler name). The Watcom C/C++ environment variable names are:

WCC used with the Watcom C compiler for 16-bit Intel platforms

Example:
```
C>set wcc=/d1 /ot
```

WPP used with the Watcom C++ compiler for 16-bit Intel platforms

Example:
```
C>set wpp=/d1 /ot
```

WCC386 used with the Watcom C compiler for 32-bit Intel platforms

Example:
```
C>set wcc386=/d1 /ot
```

WPP386 used with the Watcom C++ compiler for 32-bit Intel platforms

Example:
```
C>set wpp386=/d1 /ot
```

The options specified in environment variables are processed before options specified on the command line. The above examples define the default options to be "d1" (include line number debugging information in the object file), and "ot" (favour time optimizations over size optimizations).

Whenever you wish to specify an option that requires the use of an "=" character, you can use the "#" character in its place. This is required by the syntax of the "SET" command.

Once a particular environment variable has been defined, those options listed become the default each time the associated compiler is used. The compiler command line can be used to override any options specified in the environment string.

These environment variables are not examined by the Watcom Compile and Link utilities. Since the Watcom Compile and Link utilities pass the relevant options found in their associated environment variables to the compiler command line, their environment variable options take precedence over the options specified in the environment variables associated with the compilers.

> ***Hint:*** If you are running DOS and you use the same compiler options all the time, you may find it handy to define the environment variable in your DOS system initialization file, `AUTOEXEC.BAT`.
>
> If you are running Windows NT, use the "System" icon in the **Control Panel** to define environment variables.
>
> If you are running OS/2 and you use the same compiler options all the time, you may find it handy to define the environment variable in your OS/2 system initialization file, `CONFIG.SYS`.

3.4 Watcom C/C++ Command Line Examples

The following are some examples of using Watcom C/C++ to compile C/C++ source programs.

Example:

```
C>compiler_name report /d1 /s
```

The compiler processes `REPORT.C(PP)` producing an object file which contains source line number information. Stack overflow checking is omitted from the object code.

Example:

```
C>compiler_name /mm /fpc calc
```

The compiler compiles `CALC.C(PP)` for the Intel "medium" memory model and generates calls to floating-point library emulation routines for all floating-point operations. Memory models are described in the chapter entitled "16-bit Memory Models" on page 135.

Example:

```
C>compiler_name kwikdraw /2 /fpi87 /oaxt
```

The compiler processes `KWIKDRAW.C(PP)` producing 16-bit object code for an Intel 286 system equipped with an Intel 287 numeric data processor (or any upward compatible 386/387, 486DX, or Pentium system). While the choice of these options narrows the number of microcomputer systems where this code will execute, the resulting code will be highly optimized for this type of system.

Example:

```
C>compiler_name /mf /3s calc
```

The compiler compiles `CALC.C(PP)` for the Intel 32-bit "flat" memory model. The compiler will generate 386 instructions based on 386 instruction timings using the stack-based argument passing convention. The resulting code will be optimized for Intel 386 systems. Memory models are described in the chapter entitled "32-bit Memory Models" on page 217. Argument passing conventions are described in the chapter entitled "32-bit Assembly Language Considerations" on page 223.

Example:

```
C>compiler_name kwikdraw /4r /fpi87 /oaimxt
```

The compiler processes `KWIKDRAW.C(PP)` producing 32-bit object code for an Intel 386-compatible system equipped with a 387 numeric data processor. The compiler will generate 386 instructions based on 486 instruction timings using the register-based argument passing convention. The resulting code will be highly optimized for Intel 486 systems.

Example:

```
C>compiler_name ..\source\modabs /d2
```

The compiler processes `..\SOURCE\MODABS.C(PP)` (a file in a directory which is adjacent to the current one). The object file is placed in the current directory. Included with the object code and data is information on local symbols and data types. The code generated is straight-forward, unoptimized code which can be readily debugged with the Watcom Debugger.

Example:

```
C>set compiler_name=/i#\includes /mc
C>compiler_name \cprogs\grep.tst /fi=iomods.c
```

The compiler processes the program contained in the file `\CPROGS\GREP.TST`. The file `IOMODS.C` is included as if it formed part of the source input stream. The include search path and memory model options are defaults each time the compiler is invoked. The memory model option could be overridden on the command line. After looking for an "include" file in the current directory, the compiler will search each directory listed in the "i" path. See the section entitled "Watcom C/C++ #include File Processing" on page 85 for more information.

Example:

```
C>compiler_name grep /fo=..\obj\
```

The compiler processes the program contained in the file `GREP.C(PP)` which is located in the current directory. The object file is placed in the directory `..\OBJ` under the name `GREP.OBJ`.

Example:

```
C>compiler_name /dDBG=1 grep /fo=..\obj\.dbo
```

The compiler processes the program contained in the file `GREP.C(PP)` which is located in the current directory. The macro "DBG" is defined so that conditional debugging statements that have been placed in the source are compiled. The object file is placed in the directory `..\OBJ` and its filename extension will be ".dbo" (instead of ".obj"). Selection of a different filename extension permits easy identification of object files that have been compiled with debugging statements.

Example:

```
C>compiler_name /g=GKS /s \gks\gopks
```

The compiler generates code for `GOPKS.C(PP)` and places it into the "GKS" group. If the "g" option had not been specified, the code would not have been placed in any group. Assume that this file contains the definition of the routine `gopengks` as follows:

```
void far gopengks( int workstation, long int h )
{
    .
    .
    .
}
```

For a small code model, the routine `gopengks` must be defined in this file as `far` since it is placed in another group. The "s" option is also specified to prevent a run-time call to the stack overflow check routine which will be placed in a different code segment at link time. The `gopengks` routine must be prototyped by C routines in other groups as

```
void far gopengks( int workstation, long int h );
```

since it will appear in a different code segment.

3.5 Benchmarking Hints

The Watcom C/C++ compiler contains many options for controlling the code to be produced. It is impossible to have a certain set of compiler options that will produce the absolute fastest execution times for all possible applications. With that said, we will list the compiler options that we think will give the best execution times for most applications. You may have to experiment with different options to see which combination of options generates the fastest code for your particular application.

The recommended options for generating the fastest 16-bit Intel code are:

Pentium Pro /oneatx /oh /oi+ /ei /zp8 /6 /fpi87 /fp6

Pentium /oneatx /oh /oi+ /ei /zp8 /5 /fpi87 /fp5

486 /oneatx /oh /oi+ /ei /zp8 /4 /fpi87 /fp3

386 /oneatx /oh /oi+ /ei /zp8 /3 /fpi87 /fp3

286 /oneatx /oh /oi+ /ei /zp8 /2 /fpi87 /fp2

186 /oneatx /oh /oi+ /ei /zp8 /1 /fpi87

8086 /oneatx /oh /oi+ /ei /zp8 /0 /fpi87

The recommended options for generating the fastest 32-bit Intel code are:

Pentium Pro /oneatx /oh /oi+ /ei /zp8 /6 /fp6

Pentium /oneatx /oh /oi+ /ei /zp8 /5 /fp5

486 /oneatx /oh /oi+ /ei /zp8 /4 /fp3

386 /oneatx /oh /oi+ /ei /zp8 /3 /fp3

The "oi+" option is for C++ only. Under some circumstances, the "ob" and "ol+" optimizations may also give better performance with 32-bit Intel code.

Option "on" causes the compiler to replace floating-point divisions with multiplications by the reciprocal. This generates faster code (multiplication is faster than division), but the result may not be the same because the reciprocal may not be exactly representable.

Option "oe" causes small user written functions to be expanded in-line rather than generating a call to the function. Expanding functions in-line can further expose other optimizations that couldn't otherwise be detected if a call was generated to the function.

Option "oa" causes the compiler to relax alias checking.

Option "ot" must be specified to cause the code generator to select code sequences which are faster without any regard to the size of the code. The default is to select code sequences which strike a balance between size and speed.

Option "ox" is equivalent to "obiklmr" and "s" which causes the compiler/code generator to do branch prediction ("ob"), expand intrinsic functions in-line ("oi"), enable control flow prologues and epilogues ("ok"), perform loop optimizations ("ol"), generate 387 instructions in-line for math functions such as sin, cos, sqrt ("om"), reorder instructions to avoid pipeline stalls ("or"), and to not generate any stack overflow checking ("s"). Option "or" is very important for generating fast code for the Pentium and Pentium Pro processors.

Option "oh" causes the compiler to attempt repeated optimizations (which can result in longer compiles but more optimal code).

Option "oi+" causes the C++ compiler to expand intrinsic functions in-line (just like "oi") but also sets the *inline_depth* to its maximum (255). By default, *inline_depth* is 3. The *inline_depth* can also be changed by using the C++ `inline_depth` pragma.

Option "ei" causes the compiler to allocate at least an "int" for all enumerated types.

Option "zp8" causes all data to be aligned on 8 byte boundaries. The default is "zp2" for the 16-bit compiler and "zp8" for 32-bit compiler. If, for example, "zp1" packing was specified then this would pack all data which would reduce the amount of data memory required but would require extra clock cycles to access data that is not on an appropriate boundary.

Options "0", "1", "2", "3", "4", "5" and "6" emit Intel code sequences optimized for processor-specific instruction set features and timings. For 16-bit Intel applications, the use of these options may limit the range of systems on which the application will run but there are execution performance improvements.

Options "fp2", "fp3", "fp5" and "fp6" emit Intel floating-point operations targetted at specific features of the math coprocessor in the Intel series. For 16-bit Intel applications, the use of these options may limit the range of systems on which the application will run but there are execution performance improvements.

82 Benchmarking Hints

Option "fpi87" causes in-line Intel 80x87 numeric data processor instructions to be generated into the object code for floating-point operations. Floating-point instruction emulation is not included so as to obtain the best floating-point performance in 16-bit Intel applications.

For 32-bit Intel applications, the use of the "fp5" option will give good performance on the Intel Pentium but less than optimal performance on the 386 and 486. The use of the "5" option will give good performance on the Pentium and minimal, if any, impact on the 386 and 486. Thus, the following set of options gives good overall performance for the 386, 486 and Pentium processors.

```
/oneatx /oh /oi+ /ei /zp8 /5 /fp3
```

3.6 Compiler Diagnostics

If the compiler prints diagnostic messages to the screen, it will also place a copy of these messages in a file in your current directory. The file will have the same file name as the source file and an extension of ".err". The compiler issues two types of diagnostic messages, namely warnings or errors. A warning message does not prevent the production of an object file. However, error messages indicate that a problem is severe enough that it must be corrected before the compiler will produce an object file. The error file is a handy reference when you wish to correct the errors in the source file.

Just to illustrate the diagnostic features of Watcom C/C++, we will modify the "hello" program in such a way as to introduce some errors.

Example:

```
#include <stdio.h>

int main()
  {
     int x;
     printf( "Hello world\n" );
     return( y );
  }
```

The equivalent C++ program follows:

Example:

```
#include <iostream.h>
#include <iomanip.h>

int main()
{
    int x;
    cout << "Hello world" << endl;
    return( y );
}
```

In this example, we have added the lines:

```
int x;
```

and

```
return( y );
```

and changed the keyword `void` to `int`.

We compile the program with the "warning" option.

Example:

```
C>compiler_name hello /w3
```

For the C program, the following output appears on the screen.

```
hello.c(7): Error! E1011: Symbol 'y' has not been declared
hello.c(5): Warning! W202: Symbol 'x' has been defined, but not
                           referenced
hello.c: 8 lines, included 174, 1 warnings, 1 errors
```

For the C++ program, the following output appears on the screen.

```
hello.cpp(8): Error! E029: (col 13) symbol 'y' has not been declared
hello.cpp(9): Warning! W014: (col 1) no reference to symbol 'x'
hello.cpp(9): Note! N392: (col 1) 'int x' in 'int main( void )'
              defined in: hello.cpp(6) (col 9)
hello.cpp: 9 lines, included 1628, 1 warning, 1 error
```

Here we see an example of both types of messages. An error and a warning message have been issued. As indicated by the error message, we require a declarative statement for the identifier `y`. The warning message indicates that, while it is not a violation of the rules of C/C++ to define a variable without ever using it, we probably did not intend to do so. Upon examining the program, we find that:

1. the variable `x` should have been assigned a value, and

2. the variable `y` has probably been incorrectly typed and should have been entered as `x`.

The complete list of Watcom C/C++ diagnostic messages is presented in an appendix of this guide.

3.7 Watcom C/C++ #include File Processing

When using the `#include` preprocessor directive, a header is identified by a sequence of characters placed between the "<" and ">" delimiters (e.g., <file>) and a source file is identified by a sequence of characters enclosed by quotation marks (e.g., "file"). Watcom C/C++ makes a distinction between the use of "<>" or quotation marks to surround the name of the file to be included. The search techniques for header files and source files are slightly different. Consider the following example.

Example:

```
#include <stdio.h>  /* a system header file */
#include "stdio.h"  /* your own header or source file */
```

You should use "<" and ">" when referring to standard or system header files and quotation marks when referring to your own header and source files.

The character sequence placed between the delimiters in an `#include` directive represents the name of the file to be included. The file name may include drive, path, and extension.

It is not necessary to include the drive and path specifiers in the file specification when the file resides on a different drive or in a different directory. Watcom C/C++ provides a mechanism for looking up include files which may be located in various directories and disks of the computer system. Watcom C/C++ searches directories for header and source files in the following order (the search stops once the file has been located):

1. If the file specification enclosed in quotation marks ("file-spec") or angle brackets (<file-spec>) contains the complete drive and path specification, that file is included (provided it exists). No other searching is performed. The drive need not be specified in which case the current drive is assumed.

2. If the file specification is enclosed in quotation marks, the current directory is searched.

3. Next, if the file specification is enclosed in quotation marks, the directory of the file containing the `#include` directive is searched. If the current file is also an `#include` file, the directory of the parent file is searched next. This search

continues recursively through all the nested `#include` files until the original source file's directory is searched.

4. Next, if the file specification enclosed in quotation marks ("file-spec") or in angle brackets (<file-spec>), each directory listed in the "i" path is searched (in the order that they were specified).

5. Next, each directory listed in the **<os>_INCLUDE** environment variable is searched (in the order that they were specified). The environment variable name is constructed from the current build target name. The default build targets are:

   DOS when the host operating system is DOS,

   OS2 when the host operating system is OS/2,

   NT when the host operating system is Windows NT/95, or

   QNX when the host operating system is QNX.

   For example, the environment variable **OS2_INCLUDE** will be searched if the build target is "OS2". The build target would be OS/2 if:

   1. the host operating system is OS/2 and the "bt" option was not specified, or
   2. the "bt=OS2" option was explicitly specified.

6. Next, each directory listed in the **INCLUDE** environment variable is searched (in the order that they were specified).

7. Finally, if the file specification is enclosed in quotation marks, an adjacent "H" directory (i.e., `..\H`) is searched if it exists.

In the above example, `<stdio.h>` and `"stdio.h"` could refer to two different files if there is a `STDIO.H` in the current directory and one in the Watcom C/C++ include file directory `(\WATCOM\H)` and the current directory is not listed in an "i" path or the **INCLUDE** environment variable.

The compiler will search the directories listed in "i" paths (see description of the "i" option) and the **INCLUDE** environment variable in a manner analogous to that which the operating system shell will use when searching for programs by using the **PATH** environment variable.

The "SET" command is used to define an **INCLUDE** environment variable that contains a list of directories. A command of the form

```
SET INCLUDE=[d:]path;[d:]path...
```

is issued before running Watcom C/C++ the first time. The brackets indicate that the drive "d:" is optional and the ellipsis indicates that any number of paths may be specified. For Windows NT, use the "System" icon in the Control Panel to define environment variables.

We illustrate the use of the `#include` directive in the following example.

Example:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>

#include "common.c"

int main()
  {
    initialize();
    update_files();
    create_report();
    finalize();
  }

#include "part1.c"
#include "part2.c"
```

If the above text is stored in the source file `REPORT.C` in the current directory then we might issue the following commands to compile the application.

Example:

```
C>rem -- Two places to look for include files
C>set include=c:\watcom\h;b:\headers
C>rem -- Now compile application specifying a
C>rem    third location for include files
C>compiler_name report /fo=..\obj\ /i=..\source
```

In the above example, the "SET" command is used to define the **INCLUDE** environment variable. It specifies that the `\WATCOM\H` directory (of the "C" disk) and the `\HEADERS` directory (a directory of the "B" disk) are to be searched.

The Watcom C/C++ "i" option defines a third place to search for include files. The advantage of the **INCLUDE** environment variable is that it need not be specified each time the compiler is run.

3.8 Watcom C/C++ Preprocessor

The Watcom C/C++ preprocessor forms an integral part of Watcom C/C++. When any form of the "p" option is specified, only the preprocessor is invoked. No code is generated and no object file is produced. The output of the preprocessor is written to the standard output file, although it can also be redirected to a file using the "fo" option. Suppose the following C/C++ program is contained in the file `MSGID.C`.

Example:

```
#define _IBMPC 0
#define _IBMPS2 1

#if _TARGET == _IBMPS2
char *SysId = { "IBM PS/2" };
#else
char *SysId = { "IBM PC" };
#endif

/* Return pointer to System Identification */

char *GetSysId()
  {
    return( SysId );
  }
```

We can use the Watcom C/C++ preprocessor to generate the C/C++ code that would actually be compiled by the compiler by issuing the following command.

Example:

```
C>compiler_name msgid /plc /fo /d_TARGET=_IBMPS2
```

The file `MSGID.I` will be created and will contain the following C/C++ code.

```
#line 1 "msgid.c"



char *SysId = { "IBM PS/2" };
#line 9 "msgid.c"

/* Return pointer to System Identification */

char *GetSysId()
  {
    return( SysId );
  }
```

Note that the file `MSGID.I` can be used as input to the compiler.

Example:

```
C>compiler_name msgid.i
```

Since `#line` directives are present in the file, the compiler can issue error messages in terms of the original source file line numbers.

3.9 Watcom C/C++ Predefined Macros

In addition to the standard ANSI/ISO-defined macros supported by the Watcom C/C++ compilers, several additional system-dependent macros are also defined. These are described in this section. See the ***WATCOM C Language Reference*** manual for a description of the standard macros.

The Watcom C/C++ compilers run on various host operating systems including DOS, OS/2, Windows NT, Windows 95 and QNX. Any of the supported host operating systems can be used to develop applications for a number of target systems. By default, the target operating system for the application is the same as the host operating system unless some option or combination of options is specified. For example, DOS applications are built on DOS by default, OS/2 applications are built on OS/2 by default, and so on. But the flexibility is there to build applications for other operating systems/environments.

The macros described below may be used to identify the target system for which the application is being compiled. (Note: In several places in the following text, a pair of underscore characters appears as __ which resembles a single, elongated underscore.)

The Watcom C/C++ compilers support both 16-bit and 32-bit application development. The following macros are defined for 16-bit and 32-bit target systems.

16-bit	32-bit
__X86__	__X86__
__I86__	__386__
M_I86	M_I386
_M_I86	_M_I386
_M_IX86	_M_IX86

Notes:

1. The __X86__ identifies the target as an Intel environment.

2. The __I86__, M_I86 and _M_I86 macros identify the target as a 16-bit Intel environment.

3. The __386__, M_I386 and _M_I386 macros identify the target as a 32-bit Intel environment.

4. The _M_IX86 macro is identically equal to 100 times the architecture compiler option value (/0, /1, /2, /3, /4, /5, etc.). If "/5" (Pentium instruction timings) was specified as a compiler option, then the value of _M_IX86 would be 500.

The Watcom C/C++ compilers support application development for a variety of operating systems. The following macros are defined for particular target operating systems.

Target	Macros
DOS	__DOS__, _DOS, MSDOS
OS/2	__OS2__
QNX	__QNX__
Netware	__NETWARE__, __NETWARE_386__
NT	__NT__
Windows	__WINDOWS__, _WINDOWS, __WINDOWS_386__

Notes:

1. The __DOS__, _DOS and MSDOS macros are defined when the build target is "DOS" (16-bit DOS or 32-bit extended DOS).

2. The __OS2__ macro is defined when the build target is "OS2" (16-bit or 32-bit OS/2).

3. The __QNX__ macro is defined when the build target is "QNX" (16-bit or 32-bit QNX).

4. The __NETWARE__ and __NETWARE_386__ macros are defined when the build target is "NETWARE" (Novell NetWare).

5. The __NT__ macro is defined when the build target is "NT" (Windows NT and Windows 95).

6. The __WINDOWS__ macro is defined when the build target is "WINDOWS" or one of the "zw", "zW", "zWs" options is specified (identifies the target operating system as 16-bit Windows or 32-bit extended Windows but not Windows NT or Windows 95).

7. The _WINDOWS macro is defined when the build target is "WINDOWS" or one of the "zw", "zW", "zWs" options is specified and you are using a 16-bit compiler (identifies the target operating system as 16-bit Windows).

8. The __WINDOWS_386__ macro is defined when the build target is "WINDOWS" or the "zw" option is specified and you are using a 32-bit compiler (identifies the target operating system as 32-bit extended Windows).

The following macros are defined for the indicated options.

Option	Macro
bm	_MT
br	_DLL
fpi	__FPI__
fpi87	__FPI__
j	__CHAR_SIGNED__
oi	__INLINE_FUNCTIONS__
xr	_CPPRTTI (C++ only)
xs	_CPPUNWIND (C++ only)
xss	_CPPUNWIND (C++ only)
xst	_CPPUNWIND (C++ only)
za	NO_EXT_KEYS
zw	__WINDOWS__
zW	__WINDOWS__
zWs	__WINDOWS__

The following memory model macros are defined for the indicated memory model options.

Option	All	16-bit only		32-bit only	
mf	__FLAT__			M_386FM	_M_386FM
ms	__SMALL__	M_I86SM	_M_I86SM	M_386SM	_M_386SM
mm	__MEDIUM__	M_I86MM	_M_I86MM	M_386MM	_M_386MM
mc	__COMPACT__	M_I86CM	_M_I86CM	M_386CM	_M_386CM
ml	__LARGE__	M_I86LM	_M_I86LM	M_386LM	_M_386LM
mh	__HUGE__	M_I86HM	_M_I86HM		

The following macros indicate which compiler is compiling the C/C++ source code.

__cplusplus Watcom C++ predefines the macro `__cplusplus` to identify the compiler as a C++ compiler.

__WATCOMC__
Watcom C/C++ predefines the macro `__WATCOMC__` to identify the compiler as one of the Watcom C/C++ compilers.

The value of the macro depends on the version number of the compiler. The value is 100 times the version number (version 8.5 yields 850, version 9.0 yields 900, etc.).

__WATCOM_CPLUSPLUS__
Watcom C/C++ predefines the macro `__WATCOM_CPLUSPLUS__` to identify the compiler as one of the Watcom C++ compilers.

The value of the macro depends on the version number of the compiler. The value is 100 times the version number (version 10.0 yields 1000, version 10.5 yields 1050, etc.).

The following macros are defined for compatibility with Microsoft.

__CPPRTTI Watcom C++ predefines the `__CPPRTTI` macro to indicate that C++ Run-Time Type Information (RTTI) is in force. This macro is predefined if the Watcom C++ "xr" compile option is specified and is not defined otherwise.

__CPPUNWIND
Watcom C++ predefines the `__CPPUNWIND` macro to indicate that C++ exceptions supported. This macro is predefined if any of the Watcom C++ "xs", "xss" or "xst" compile options are specified and is not defined otherwise.

_fastcall, __fastcall
Watcom C++ predefines the `_fastcall` and `__fastcall` macros to an

empty string so that source code containing this Microsoft keyword can be compiled without syntax errors. The `_fastcall` keyword refers to a Microsoft calling convention that is not supported by Watcom C/C++. Watcom's calling conventions are already "fast" to start with; hence, the keyword is not required.

_inline, __inline

Watcom C++ predefines the `_inline` and `__inline` macros to be identical to the `inline` keyword.

_INTEGRAL_MAX_BITS

&prodname predefines the `_INTEGRAL_MAX_BITS` macro to indicate that maximum number of bits supported in an integral type (see the description of the "__int64" keyword in the next section). Its value is 64 currently.

_PUSHPOP_SUPPORTED

Watcom C/C++ predefines the `_PUSHPOP_SUPPORTED` macro to indicate that `#pragma pack(push)` and `#pragma pack(pop)` are supported.

_STDCALL_SUPPORTED

Watcom C/C++ predefines the `_STDCALL_SUPPORTED` macro to indicate that the standard 32-bit Win32 calling convention is supported.

The following table summarizes the predefined macros supported by the compilers and the values that the respective compilers assign to them. A "yes" under the column means that the compiler supports the macro with the indicated value. Note that the C and C++ compilers sometime support the same macro but with different values (including no value which means the symbol is defined without a value).

Predefined Macro and Setting	Supported by Compiler wcc	wcc386	wpp	wpp386
__386__=1		Yes		Yes
__3R__=1				Yes
_based=__based	Yes	Yes	Yes	Yes
_cdecl=__cdecl	Yes	Yes	Yes	Yes
cdecl=__cdecl	Yes	Yes	Yes	Yes
__cplusplus=1			Yes	Yes
_CPPRTTI=1			Yes	Yes
_CPPUNWIND=1			Yes	Yes
_export=__export	Yes	Yes	Yes	Yes
_far16=__far16	Yes	Yes	Yes	Yes
_far=__far	Yes	Yes	Yes	Yes
far=__far	Yes	Yes	Yes	Yes
__fastcall=			Yes	Yes
_fastcall=			Yes	Yes
_fastcall=__fastcall	Yes	Yes		
__FLAT__=1		Yes		Yes
_fortran=__fortran	Yes	Yes	Yes	Yes
fortran=__fortran	Yes	Yes	Yes	Yes
__FPI__=1	Yes	Yes	Yes	Yes
_huge=__huge	Yes	Yes	Yes	Yes
huge=__huge	Yes	Yes	Yes	Yes
__I86__=1	Yes		Yes	
_inline=__inline	Yes	Yes		
__inline=inline			Yes	Yes
_inline=inline			Yes	Yes
_INTEGRAL_MAX_BITS=64	Yes	Yes	Yes	Yes
_interrupt=__interrupt	Yes	Yes	Yes	Yes
interrupt=__interrupt	Yes	Yes	Yes	Yes
_loadds=__loadds	Yes	Yes	Yes	Yes
_M_386FM=1				Yes
M_386FM=1				Yes
_M_I386=1				Yes
M_I386=1		Yes		Yes
_M_I86=1			Yes	
M_I86=1	Yes		Yes	
_M_I86SM=1	Yes		Yes	
M_I86SM=1	Yes		Yes	
_M_IX86=0	Yes		Yes	
_M_IX86=500		Yes		Yes
_near=__near	Yes	Yes	Yes	Yes
near=__near	Yes	Yes	Yes	Yes
__NT__=1 (on Win32 platform)	Yes	Yes	Yes	Yes
_pascal=__pascal	Yes	Yes	Yes	Yes
pascal=__pascal	Yes	Yes	Yes	Yes
_saveregs=__saveregs	Yes	Yes	Yes	Yes
_segment=__segment	Yes	Yes	Yes	Yes
_segname=__segname	Yes	Yes	Yes	Yes
_self=__self	Yes	Yes	Yes	Yes
__SMALL__=1	Yes		Yes	
SOMDLINK=__far	Yes			
SOMDLINK=_Syscall		Yes		Yes
SOMLINK=__cdecl	Yes			
SOMLINK=_Syscall		Yes		Yes
_STDCALL_SUPPORTED=1		Yes		Yes
__SW_0=1	Yes		Yes	

__SW_3R=1		Yes		Yes
__SW_5=1		Yes		Yes
__SW_FP287=1			Yes	
__SW_FP2=1	Yes			
__SW_FP387=1				Yes
__SW_FP3=1		Yes		
__SW_FPI=1	Yes	Yes	Yes	Yes
__SW_MF=1		Yes		Yes
__SW_MS=1	Yes			
__SW_ZDP=1	Yes	Yes	Yes	Yes
__SW_ZFP=1	Yes	Yes	Yes	Yes
__SW_ZGF=1		Yes		Yes
__SW_ZGP=1	Yes		Yes	
_syscall=_Syscall	Yes	Yes	Yes	Yes
_System=_Syscall	Yes	Yes	Yes	Yes
__WATCOM_CPLUSPLUS__=1100			Yes	Yes
__WATCOMC__=1100	Yes	Yes	Yes	Yes
__X86__=1	Yes	Yes	Yes	Yes

3.10 Watcom C/C++ Extended Keywords

Watcom C/C++ supports the use of some special keywords to describe system dependent attributes of functions and other object names. These attributes are inspired by the Intel processor architecture and the plethora of function calling conventions in use by compilers for this architecture. In keeping with the ANSI/ISO C and C++ language standards, Watcom C/C++ uses the double underscore (i.e., "__") or single underscore followed by uppercase letter (e.g., "_S") prefix with these keywords. To support compatibility with other C/C++ compilers, alternate forms of these keywords are also supported through predefined macros.

__near Watcom C/C++ supports the `__near` keyword to describe functions and other object names that are in near memory and pointers to near objects.

Watcom C/C++ predefines the macros `near` and `_near` to be equivalent to the `__near` keyword.

__far Watcom C/C++ supports the `__far` keyword to describe functions and other object names that are in far memory and pointers to far objects.

Watcom C/C++ predefines the macros `far, _far` and `SOMDLINK` (16-bit only) to be equivalent to the `__far` keyword.

__huge Watcom C/C++ supports the `__huge` keyword to describe functions and other object names that are in huge memory and pointers to huge objects. The 32-bit compilers treat these as equivalent to far objects.

Watcom C/C++ predefines the macros `huge` and `_huge` to be equivalent to the `__huge` keyword.

__based Watcom C/C++ supports the `__based` keyword to describe pointers to objects that appear in other segments or the objects themselves. See the section entitled "Based Pointers" on page 104 for an explanation of the `__based` keyword.

Watcom C/C++ predefines the macro `_based` to be equivalent to the `__based` keyword.

__segment Watcom C/C++ supports the `__segment` keyword which is used when describing objects of type segment. See the section entitled "Based Pointers" on page 104 for an explanation of the `__segment` keyword.

Watcom C/C++ predefines the macro `_segment` to be equivalent to the `__segment` keyword.

__segname Watcom C/C++ supports the `__segname` keyword which is used when describing segname constant based pointers or objects. See the section entitled "Based Pointers" on page 104 for an explanation of the `__segname` keyword.

Watcom C/C++ predefines the macro `_segname` to be equivalent to the `__segname` keyword.

__self Watcom C/C++ supports the `__self` keyword which is used when describing self based pointers. See the section entitled "Based Pointers" on page 104 for an explanation of the `__self` keyword.

Watcom C/C++ predefines the macro `_self` to be equivalent to the `__self` keyword.

_Packed Watcom C/C++ supports the `_Packed` keyword which is used when describing a structure. If specified before the ***struct*** keyword, the compiler will force the structure to be packed (no alignment, no gaps) regardless of the setting of the command-line option or the ***#pragma*** controlling the alignment of members.

__cdecl Watcom C/C++ supports the `__cdecl` keyword to describe C functions that are called using a special convention.

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine returns a pointer in register AX/EAX to the return value which is stored in the data segment (DGROUP).

4. For the 16-bit compiler, registers AX, BX, CX and DX, and segment register ES are not saved and restored when a call is made.

5. For the 32-bit compiler, registers EAX, ECX and EDX are not saved and restored when a call is made.

Watcom C/C++ predefines the macros `cdecl`, `_cdecl`, `_Cdecl` and `SOMLINK` (16-bit only) to be equivalent to the `__cdecl` keyword.

__pascal Watcom C/C++ supports the `__pascal` keyword to describe Pascal functions that are called using a special convention described by a pragma in the "stddef.h" header file.

Watcom C/C++ predefines the macros `pascal`, `_pascal` and `_Pascal` to be equivalent to the `__pascal` keyword.

__fortran Watcom C/C++ supports the `__fortran` keyword to describe functions that are called from FORTRAN. It converts the name to uppercase letters and suppresses the "_" which is appended to the function name for certain calling conventions.

Watcom C/C++ predefines the macros `fortran` and `_fortran` to be equivalent to the `__fortran` keyword.

__interrupt Watcom C/C++ supports the `__interrupt` keyword to describe a function that is an interrupt handler.

Example:

```
#include <i86.h>

void __interrupt int10( union INTPACK r )
{
        .
        .
        .
}
```

The code generator will emit instructions to save all registers. The registers are saved on the stack in a specific order so that they may be referenced using the

"INTPACK" union as shown in the DOS example above. The code generator will emit instructions to establish addressability to the program's data segment since the DS segment register contents are unpredictable. The function will return using an "IRET" (16-bit) or "IRETD" (32-bit) (interrupt return) instruction.

Watcom C/C++ predefines the macros `interrupt` and `_interrupt` to be equivalent to the `__interrupt` keyword.

__declspec(modifier)

Watcom C/C++ supports the `__declspec` keyword for compatibility with Microsoft C++. The `__declspec` keyword is used to modify storage-class attributes of functions and/or data. There are several modifiers that can be specified with the `__declspec` keyword: `thread`, `naked`, `dllimport`, `dllexport`, `__pragma( "string" )`, `__cdecl`, `__pascal`, `__fortran`, `__stdcall`, and `__syscall`. These attributes are a property only of the declaration of the object or function to which they are applied. Unlike the `__near` and `__far` keywords, which actually affect the type of object or function (in this case, 2- and 4-byte addresses), these storage-class attributes do not redefine the type attributes of the object itself. The `__pragma` modifier is supported by Watcom C++ only. The `thread` attribute affects data and objects only. The `naked`, `__pragma`, `__cdecl`, `__pascal`, `__fortran`, `__stdcall`, and `__syscall` attributes affect functions only. The `dllimport` and `dllexport` attributes affect functions, data, and objects. For more information on the `__declspec` keyword, please see the section entitled "The __declspec Keyword" on page 109.

__export Watcom C/C++ supports the `__export` keyword to describe functions and other object names that are to be exported from a Microsoft Windows DLL, OS/2 DLL, or Netware NLM. See also the description of the "zu" option.

Example:

```
void __export _Setcolor( int color )
{
        .
        .
        .
}
```

Watcom C/C++ predefines the macro `_export` to be equivalent to the `__export` keyword.

__loadds Watcom C/C++ supports the `__loadds` keyword to describe functions that require specific loading of the DS register to establish addressability to the

function's data segment. This keyword is useful in describing a function that will be placed in a Microsoft Windows or OS/2 1.x Dynamic Link Library (DLL). See also the description of the "nd" and "zu" options.

Example:

```
void __export __loadds _Setcolor( int color )
{
        .
        .
        .
}
```

If the function in an OS/2 1.x Dynamic Link Library requires access to private data, the data segment register must be loaded with an appropriate value since it will contain the DS value of the calling application upon entry to the function.

Watcom C/C++ predefines the macro `_loadds` to be equivalent to the `__loadds` keyword.

__saveregs Watcom C/C++ recognizes the `__saveregs` keyword which is an attribute used by C/C++ compilers to describe a function that must save and restore all registers.

Watcom C/C++ predefines the macro `_saveregs` to be equivalent to the `__saveregs` keyword.

__stdcall (32-bit only) The `__stdcall` keyword may be used with function definitions, and indicates that the 32-bit Win32 calling convention is to be used.

Notes:

1. All symbols are preceded by an underscore character.

2. All C symbols (extern "C" symbols in C++) are suffixed by "@nnn" where "nnn" is the sum of the argument sizes (each size is rounded up to a multiple of 4 bytes so that char and short are size 4). When the argument list contains "...", the "@nnn" suffix is omitted.

3. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The called routine will remove the arguments from the stack.

4. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack

immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

5. Registers EAX, ECX and EDX are not saved and restored when a call is made.

__syscall (32-bit only) The `__syscall` keyword may be used with function definitions, and indicates that the calling convention used is compatible with functions provided by 32-bit OS/2.

Notes:

1. Symbols names are not modified, that is, they are not adorned with leading or trailing underscores.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

4. Registers EAX, ECX and EDX are not saved and restored when a call is made.

Watcom C/C++ predefines the macros `_syscall, _System, SOMLINK` (32-bit only) and `SOMDLINK` (32-bit only) to be equivalent to the `__syscall` keyword.

__far16 (32-bit only) Watcom C/C++ recognizes the `__far16` keyword which can be used to define far 16-bit (far16) pointers (16-bit selector with 16-bit offset) or far 16-bit function prototypes. This keyword can be used under 32-bit OS/2 to call 16-bit functions from your 32-bit flat model program. Integer arguments will automatically be converted to 16-bit integers, and 32-bit pointers will be converted to far16 pointers before calling a special thunking layer to transfer control to the 16-bit function.

Watcom C/C++ predefines the macros `_far16` and `_Far16` to be equivalent to the `__far16` keyword. This keyword is compatible with Microsoft C.

In the OS/2 operating system (version 2.0 or higher), the first 512 megabytes of the 4 gigabyte segment referenced by the DS register is divided into 8192 areas of 64K bytes each. A far16 pointer consists of a 16-bit selector referring to one of the 64K byte areas, and a 16-bit offset into that area.

A pointer declared as,

[type] __far16 *name;

defines an object that is a far16 pointer. If such a pointer is accessed in the 32-bit environment, the compiler will generate the necessary code to convert between the far16 pointer and a "flat" 32-bit pointer.

For example, the declaration,

```
char __far16 *bufptr;
```

declares the object `bufptr` to be a far16 pointer to ***char.***

A function declared as,

[type] __far16 func([arg_list]);

declares a 16-bit function. Any calls to such a function from the 32-bit environment will cause the compiler to convert any 32-bit pointer arguments to far16 pointers, and any ***int*** arguments from 32 bits to 16 bits. (In the 16-bit environment, an object of type ***int*** is only 16 bits.) Any return value from the function will have its return value converted in an appropriate manner.

For example, the declaration,

```
char * __far16 Scan( char *buffer, int len, short err );
```

declares the 16-bit function `Scan.` When this function is called from the 32-bit environment, the `buffer` argument will be converted from a flat 32-bit pointer to a far16 pointer (which, in the 16-bit environment, would be declared as `char __far *.` The `len` argument will be converted from a 32-bit integer to a 16-bit integer. The `err` argument will be passed unchanged. Upon returning, the far16 pointer (far pointer in the 16-bit environment) will be converted to a 32-bit pointer which describes the equivalent location in the 32-bit address space.

_Seg16 (32-bit only) Watcom C/C++ recognizes the `_Seg16` keyword which has a similar but not identical function as the `__far16` keyword described above. This keyword is compatible with IBM C Set/2.

In the OS/2 operating system (version 2.0 or higher), the first 512 megabytes of the 4 gigabyte segment referenced by the DS register is divided into 8192 areas of 64K bytes each. A far16 pointer consists of a 16-bit selector referring to one of the 64K byte areas, and a 16-bit offset into that area.

Note that `_Seg16` is **not** interchangeable with `__far16.`

A pointer declared as,

[type] * _Seg16 name;

defines an object that is a far16 pointer. Note that the `_Seg16` appears on the right side of the `*` which is opposite to the `__far16` keyword described above.

For example,

```
char * _Seg16 bufptr;
```

declares the object `bufptr` to be a far16 pointer to ***char*** (the same as above).

The `_Seg16` keyword may not be used to describe a 16-bit function. A ***#pragma*** directive must be used instead. A function declared as,

[type] * _Seg16 func([parm_list]);

declares a 32-bit function that returns a far16 pointer.

For example, the declaration,

```
char * _Seg16 Scan( char * buffer, int len, short err );
```

declares the 32-bit function `Scan.` No conversion of the argument list will take place. The return value is a far16 pointer.

__pragma Watcom C++ supports the `__pragma` keyword to support in-lining of member functions. The `__pragma` keyword must be followed by parentheses containing a string that names an auxiliary pragma. Here is a simplified example showing usage and syntax.

Example:

```
#pragma aux fast_mul = \
    "imul eax,edx" \
    parm caller [eax] [edx] \
    value struct;

struct fixed {
    unsigned v;
};

fixed __pragma( "fast_mul") operator *( fixed,
fixed );

fixed two = { 2 };
fixed three = { 3 };

fixed foo()
{
    return two * three;
}
```

See the chapters entitled "16-bit Pragmas" on page 161 and "32-bit Pragmas" on page 247 for more information on pragmas.

__int64 Watcom C/C++ supports the `__int64` keyword to define 64-bit integer data objects.

Example:

```
static __int64 bigInt;
```

Also supported are signed and unsigned 64-bit integer constants.

signed __int64 Use the "i64" suffix for a signed 64-bit integer constant.

Example:

```
12345i64
12345I64
```

unsigned __int64 Use the "ui64" suffix for a signed 64-bit integer constant.

Example:

```
12345Ui64
12345uI64
```

The run-time library supports formatting of `__int64` items (see the description of the `printf` library function).

Example:

```
#include <stdio.h>
#include <limits.h>

void main()
{
    __int64 bigint;
    __int64 bigint2;

    bigint2 = 8I64 * (LONG_MAX + 1I64);
    for( bigint = 0;
         bigint <= bigint2;
         bigint += bigint2 / 16 ) {
        printf( "Hello world %Ld\n", bigint );
    }
}
```

Restrictions

switch An `__int64` expression cannot be used in a *switch* statement.

bit fields More than 32 bits in a 64-bit bitfield is not supported.

3.11 Based Pointers

Near pointers are generally the most efficient type of pointer because they are small, and the compiler can assume knowledge about what segment of the computer's memory the pointer (offset) refers to. Far pointers are the most flexible because they allow the programmer to access any part of the computer's memory, without limitation to a particular segment. However, far pointers are bigger and slower because of the additional flexibility.

Based pointers are a compromise between the efficiency of near pointers and the flexibility of far pointers. With based pointers, the programmer takes responsibility to tell the compiler which segment a near pointer (offset) belongs to, but may still access segments of the computer's memory outside of the normal data segment (DGROUP). The result is a pointer

type which is as small as and almost as efficient as a near pointer, but with most of the flexibility of a far pointer.

An object declared as a based pointer falls into one of the following categories:

- the based pointer is in the segment described by another object,
- the based pointer, used as a pointer to another object of the same type (as in a linked list), refers to the same segment,
- the based pointer is an offset to no particular segment, and must be combined explicitly with a segment value to produce a valid pointer.

To support based pointers, the following keywords are provided:

```
__based
__segment
__segname
__self
```

The following operator is also provided:

```
:>
```

These keywords and operator are described in the following sections.

Two macros, defined in `MALLOC.H`, are also provided:

```
_NULLSEG
_NULLOFF
```

They are used in a manner similar to `NULL`, but are used with objects declared as `__segment` and `__based` respectively.

3.11.1 Segment Constant Based Pointers and Objects

A segment constant based pointer or object has its segment value based on a specific, named segment. A segment constant based object is specified as:

[type] __based(__segname("segment")) object_name;

and a segment constant based pointer is specified as:

[type] __based(__segname("segment")) *object-name;

where `segment` is the name of the segment in which the pointer or object is based. As shown above, the segment name is always specified as a string. There are three special segment names recognized by the compiler:

```
"_CODE"
"_CONST"
"_DATA"
```

The `"_CODE"` segment is the default code segment. The `"_CONST"` segment is the segment containing constant values. The `"_DATA"` segment is the default data segment. If the segment name is not one of the three recognized names, then a segment will be created with that name. If a segment constant based object is being defined, then it will be placed in the named segment. If a segment constant based pointer is being defined, then it can point at objects in the named segment.

The following examples illustrate segment constant based pointers and objects.

Example:

```
int __based( __segname( "_CODE" ) )  ival = 3;
int __based( __segname( "_CODE" ) ) *iptr;
```

`ival` is an object that resides in the default code segment. `iptr` is an object that resides in the data segment (the usual place for data objects), but points at an integer which resides in the default code segment. `iptr` is suitable for pointing at `ival.`

Example:

```
char __based( __segname( "GOODTHINGS" ) ) thing;
```

`thing` is an object which resides in the segment `GOODTHINGS,` which will be created if it does not already exist. (The creation of segments is done by the linker, and is a method of grouping objects and functions. Nothing is implicitly created during the execution of the program.)

3.11.2 Segment Object Based Pointers

A segment object based pointer derives its segment value from another named object. A segment object based pointer is specified as follows:

[type] __based(segment) *name;

where `segment` is an object defined as type `__segment.`

An object of type `__segment` may contain a segment value. Such an object is particularly designed for use with segment object based pointers.

The following example illustrates a segment object based pointer:

Example:

```
__segment            seg;
char __based( seg ) *cptr;
```

The object `seg` contains only a segment value. Whenever the object `cptr` is used to point to a character, the actual pointer value will be made up of the segment value found in `seg` and the offset value found in `cptr.` The object `seg` might be assigned values such as the following:

- a constant value (e.g., the segment containing screen memory),
- the result of the library function ***_bheapseg,***
- the segment portion of another pointer value, by casting it to the type `__segment.`

3.11.3 Void Based Pointers

A void based pointer must be explicitly combined with a segment value to produce a reference to a memory location. A void based pointer does not infer its segment value from another object. The `:>` (base) operator is used to combine a segment value and a void based pointer.

For example, on a personal computer running DOS with a color monitor, the screen memory begins at segment 0xB800, offset 0. In a video text mode, to examine the first character currently displayed on the screen, the following code could be used:

Example:

```
extern void main()
{
    __segment             screen;
    char __based( void ) *scrptr;

    screen = 0xB800;
    scrptr = 0;
    printf( "Top left character is '%c'.\n",
            *(screen:>scrptr) );
}
```

The general form of the `:>` operator is:

segment :> offset

where `segment` is an expression of type `__segment`, and `offset` is an expression of type `__based( void ) *`.

3.11.4 Self Based Pointers

A self based pointer infers its segment value from itself. It is particularly useful for structures such as linked lists, where all of the list elements are in the same segment. A self based pointer pointing to one element may be used to access the next element, and the compiler will use the same segment as the original pointer.

The following example illustrates a function which will print the values stored in the last two members of a linked list:

Example:

```
struct a {
    struct a __based( __self ) *next;
    int                         number;
};


extern void PrintLastTwo( struct a far *list )
{
  __segment                seg;
  struct a __based( seg ) *aptr;

  seg  = FP_SEG( list );
  aptr = FP_OFF( list );
  for( ; aptr != _NULLOFF; aptr = aptr->next ) {
    if( aptr->next == _NULLOFF ) {
      printf( "Last item is %d\n",
              aptr->number );
    } else if( aptr->next->next == _NULLOFF ) {
      printf( "Second last item is %d\n",
              aptr->number );
    }
  }
}
```

The argument to the function `PrintLastTwo` is a far pointer, pointing to a linked list structure anywhere in memory. It is assumed that all members of a particular linked list of this type reside in the same segment of the computer's memory. (Another instance of the linked list might reside entirely in a different segment.) The object `seg` is given the segment portion of the far pointer. The object `aptr` is given the offset portion, and is described as being based in the segment stored in `seg`.

The expression `aptr->next` refers to the `next` member of the structure stored in memory at the offset stored in `aptr` and the segment implied by `aptr,` which is the value stored in `seg.` So far, the behavior is no different than if `next` had been declared as,

```
struct a *next;
```

The expression `aptr->next->next` illustrates the difference of using a self based pointer. The first part of the expression (`aptr->next`) occurs as described above. However, using the result to point to the next member occurs by using the offset value found in the `next` member and combining it with the segment value of the *pointer used to get to that member*, which is still the segment implied by `aptr,` which is the value stored in `seg.` If `next` had not been declared using `__based( __self ),` then the second pointing operation would refer to the offset value found in the `next` member, but with the default data segment (DGROUP), which may or may not be the same segment as stored in `seg.`

3.12 The __declspec Keyword

Watcom C/C++ supports the `__declspec` keyword for compatibility with Microsoft C++. The `__declspec` keyword is used to modify storage-class attributes of functions and/or data.

__declspec(thread) is used to define thread local storage (TLS). TLS is the mechanism by which each thread in a multithreaded process allocates storage for thread-specific data. In standard multithreaded programs, data is shared among all threads of a given process, whereas thread local storage is the mechanism for allocating per-thread data.

Example:

```
__declspec(thread) static int tls_data = 0;
```

The following rules apply to the use of the `thread` attribute.

- The `thread` attribute can be used with data and objects only.
- You can specify the `thread` attribute only on data items with static storage duration. This includes global data objects (both `static` and `extern`), local static objects, and static data members of classes. Automatic data objects cannot be declared with the `thread` attribute. The following example illustrates this error:

Example:

```
#define TLS __declspec( thread )
void func1()
{
    TLS int tls_data;           // Wrong!
}

int func2( TLS int tls_data )   // Wrong!
{
    return tls_data;
}
```

- The `thread` attribute must be used for both the declaration and the definition of a thread local object, whether the declaration and definition occur in the same file or separate files. The following example illustrates this error:

Example:

```
#define TLS __declspec( thread )
extern int tls_data;    // This generates an
error, because the
TLS    int tls_data;    // declaration and the
definition differ.
```

- Classes cannot use the `thread` attribute. However, you can instantiate class objects with the `thread` attribute, as long as the objects do not need to be constructed or destructed. For example, the following code generates an error:

Example:

```
#define TLS __declspec( thread )
TLS class A     // Wrong! Classes are not
objects
{
    // Code
};
A AObject;
```

Because the declaration of objects that use the `thread` attribute is permitted, these two examples are semantically equivalent:

Example:

```
#define TLS __declspec( thread )
TLS class B
{
    // Code
} BObject;      // Okay! BObject declared
thread local.

class C
{
    // Code
};
TLS C CObject;  // Okay! CObject declared
thread local.
```

- Standard C permits initialization of an object or variable with an expression involving a reference to itself, but only for objects of non-static extent. Although C++ normally permits such dynamic initialization of an object with an expression involving a reference to itself, this type of initialization is not permitted with thread local objects.

Example:

```
#define TLS  __declspec( thread )
TLS int tls_i = tls_i;            // C and C++
error
int j = j;                        // Okay in
C++; C error
TLS int tls_k = sizeof( tls_k );  // Okay in C
and C++
```

Note that a `sizeof` expression that includes the object being initialized does not constitute a reference to itself and is allowed in C and C++.

__declspec(naked) indicates to the code generator that no prologue or epilogue sequence is to be generated for a function. Any statements other than "_asm" directives or auxiliary pragmas are not compiled. ***_asm*** Essentially, the compiler will emit a "label" with the specified function name into the code.

Example:

```
#include <stdio.h>

int __declspec( naked ) foo( int x )
{
    _asm {
#if defined(__386__)
        inc eax
#else
        inc ax
#endif
        ret
    }
}

void main()
{
    printf( "%d\n", foo( 1 ) );
}
```

The following rules apply to the use of the `naked` attribute.

- The `naked` attribute cannot be used in a data declaration. The following declaration would be flagged in error.

Example:

```
__declspec(naked) static int data_object = 0;
```

__declspec(dllimport) is used to declare functions, data and objects imported from a DLL.

Example:

```
#define DLLImport __declspec(dllimport)

DLLImport void dll_func();
DLLImport int  dll_data;
```

Functions, data and objects are exported from a DLL by use of `__declspec(export)`, the `__export` keyword (for which `__declspec(export)` is the replacement), or through linker "EXPORT" directives.

__declspec(dllexport) is used to declare functions, data and objects exported from a DLL. Declaring functions as `dllexport` eliminates the need for linker "EXPORT" directives. The `__declspec(dllexport)` attribute is a replacement for the `__export` keyword.

__declspec(__pragma("string")) is used to declare functions which adhere to the conventions described by the pragma identified by "string".

Example:

```
#include <stdio.h>

#pragma aux my_stdcall "_*" \
        parm routine [] \
        value struct struct caller [] \
        modify [eax ecx edx];

struct list {
    struct list *next;
    int         value;
    float       flt_value;
};

#define STDCALL __declspec( __pragma("my_stdcall")
)

STDCALL struct list foo( int x, char *y, double z
);

void main()
{
    int a = 1;
    char *b = "Hello there";
    double c = 3.1415926;
    struct list t;

    t = foo( a, b, c );
    printf( "%d\n", t.value );
}


struct list foo( int x, char *y, double z )
{
    struct list tmp;

    printf( "%s\n", y );
    tmp.next = NULL;
    tmp.value = x;
    tmp.flt_value = z;
    return( tmp );
}
```

The `__pragma` modifier is supported by Watcom C++ only.

__declspec(__cdecl) is used to declare functions which conform to the Microsoft compiler calling convention.

__declspec(__pascal) is used to declare functions which conform to the OS/2 1.x and Windows 3.x calling convention.

__declspec(__fortran) is used to declare functions which conform to the ***__fortran*** calling convention.

Example:

```
#include <stdio.h>

#define DLLFunc __declspec(dllimport __fortran)
#define DLLData __declspec(dllimport)

#ifdef __cplusplus
extern "C" {
#endif

DLLFunc int  dll_func( int, int, int );
DLLData int  dll_data;

#ifdef __cplusplus
};
#endif

void main()
{
  printf( "%d %d\n", dll_func( 1,2,3 ), dll_data
);
}
```

__declspec(__stdcall) is used to declare functions which conform to the 32-bit Win32 "standard" calling convention.

Example:

```
#include <stdio.h>

#define DLLFunc __declspec(dllimport _stdcall)
#define DLLData __declspec(dllimport)

DLLFunc int  dll_func( int, int, int );
DLLData int  dll_data;

void main()
{
  printf( "%d %d\n", dll_func( 1,2,3 ), dll_data
);
}
```

__declspec(__syscall) is used to declare functions which conform to the 32-bit OS/2 ***__syscall*** calling convention.

3.13 The Watcom Code Generator

The Watcom Code Generator performs such optimizations as common subexpression elimination, global flow analysis, and so on.

In some cases, the code generator can do a better job of optimizing code if it could utilize more memory. This is indicated when a

```
Not enough memory to optimize procedure 'xxxx'
```

message appears on the screen as the source program is compiled. In such an event, you may wish to make more memory available to the code generator.

A special environment variable may be used to obtain memory usage information or set memory usage limits on the code generator. The **WCGMEMORY** environment variable may be used to request a report of the amount of memory used by the compiler's code generator for its work area.

Example:

```
C>set WCGMEMORY=?
```

When the memory amount is "?" then the code generator will report how much memory was used to generate the code.

It may also be used to instruct the compiler's code generator to allocate a fixed amount of memory for a work area.

Example:

```
C>set WCGMEMORY=128
```

When the memory amount is "nnn" then exactly "nnnK" bytes will be used. In the above example, 128K bytes is requested. If less than "nnnK" is available then the compiler will quit with a fatal error message. If more than "nnnK" is available then only "nnnK" will be used.

There are two reasons why this second feature may be quite useful. In general, the more memory available to the code generator, the more optimal code it will generate. Thus, for two personal computers with different amounts of memory, the code generator may produce different (although correct) object code. If you have a software quality assurance requirement that the same results (i.e., code) be produced on two different machines then you should use this feature. To generate identical code on two personal computers with different memory configurations, you must ensure that the **WCGMEMORY** environment variable is set identically on both machines.

The second reason where this feature is useful is on virtual memory paging systems (e.g., OS/2) where an unlimited amount of memory can be used by the code generator. If a very large module is being compiled, it may take a very long time to compile it. The code generator will continue to allocate more and more memory and cause an excessive amount of paging. By restricting the amount of memory that the code generator can use, you can reduce the amount of time required to compile a routine.

4 Precompiled Headers

4.1 Using Precompiled Headers

Watcom C/C++ supports the use of precompiled headers to decrease the time required to compile several source files that include the same header file.

4.2 When to Precompile Header Files

Using precompiled headers reduces compilation time when:

- You always use a large body of code that changes infrequently.
- Your program comprises multiple modules, all of which use the same first include file and the same compilation options. In this case, the first include file along with all the files that it includes can be precompiled into one precompiled header.

Because the compiler only uses the first include file to create a precompiled header, you may want to create a master or global header file that includes all the other header files that you wish to have precompiled. Then all source files should include this master header file as the first `#include` in the source file. Even if you don't use a master header file, you can benefit from using precompiled headers for Windows programs by using `#include <windows.h>` as the first include file, or by using `#include <afxwin.h>` as the first include file for MFC applications.

The first compilation — the one that creates the precompiled header file — takes a bit longer than subsequent compilations. Subsequent compilations can proceed more quickly by including the precompiled header.

You can precompile C and C++ programs. In C++ programming, it is common practice to separate class interface information into header files which can later be included in programs that use the class. By precompiling these headers, you can reduce the time a program takes to compile.

> ***Note:*** Although you can use only one precompiled header (`.PCH`) file per source file, you can use multiple `.PCH` files in a project.

4.3 Creating and Using Precompiled Headers

Precompiled code is stored in a file called a precompiled header when you use the precompiled header option (**/fh** or **/fhq**) on the command line. The **/fh** option causes the compiler to either create a precompiled header or use the precompiled header if it already exists. The **/fhq** option is similar but prevents the compiler from issuing informational or warning messages about precompiled header files. The default name of the precompiled header file is one of `WCC.PCH, WCC386.PCH, WPP.PCH,` or `WPP386.PCH` (depending on the compiler used). You can also control the name of the precompiled header that is created or used with the **/fh=filename** or **/fhq=filename** ("specify precompiled header filename") options.

Example:

```
/fh=projectx.pch
/fhq=projectx.pch
```

4.4 The "/fh[q]" (Precompiled Header) Option

The **/fh** option instructs the compiler to use a precompiled header file with a default name of `WCC.PCH, WCC386.PCH, WPP.PCH,` or `WPP386.PCH` (depending on the compiler used) if it exists or to create one if it does not. The file is created in the current directory. You can use the **/fh=filename** option to change the default name (and placement) of the precompiled header. Add the letter "q" (for "quiet") to the option name to prevent the compiler from displaying precompiled header activity information.

The following command line uses the **/fh** option to create a precompiled header.

Example:

```
wpp /fh myprog.cpp
wpp386 /fh myprog.cpp
```

The following command line creates a precompiled header named `MYPROG.PCH` and places it in the `\PROJPCH` directory.

Example:

```
wpp /fh=\projpch\myprog.pch myprog.cpp
wpp386 /fh=\projpch\myprog.pch myprog.cpp
```

The precompiled header is created and/or used when the compiler encounters the first `#include` directive that occurs in the source file. In a subsequent compilation, the compiler performs a consistency check to see if it can use an existing precompiled header. If the consistency check fails then the compiler discards the existing precompiled header and builds a new one.

The **/fhq** form of the precompiled header option prevents the compiler from issuing warning or informational messages about precompiled header files. For example, if you change a header file, the compiler will tell you that it changed and that it must regenerate the precompiled header file. If you specify **/fhq** then the compiler just generates the new precompiled header file without displaying a message.

4.5 Consistency Rules for Precompiled Headers

If a precompiled header file exists (either the default file or one specified by **/fh=filename**), it is compared to the current compilation for consistency. A new precompiled header file is created and the new file overwrites the old unless the following requirements are met:

- The current compiler options must match those specified when the precompiled header was created.
- The current working directory must match that specified when the precompiled header was created.
- The name of the first `#include` directive must match the one that was specified when the precompiled header was created.
- All macros defined prior to the first `#include` directive must have the same values as the macros defined when the precompiled header was created. A sequence of `#define` directives need not occur in exactly the same order because there are no semantic order dependencies for `#define` directives.
- The value and order of include paths specified on the command line with **/i** options must match those specified when the precompiled header was created.
- The time stamps of all the header files (all files specified with `#include` directives) used to build the precompiled header must match those that existed when the precompiled header was created.

5 The Watcom C/C++ Libraries

The Watcom C/C++ library routines are described in the ***Watcom C Library Reference*** manual, and the ***Watcom C++ Class Library Reference*** manual.

5.1 Watcom C/C++ Library Directory Structure

Since Watcom C/C++ supports both 16-bit and 32-bit application development, libraries are grouped under two major subdirectories. The `LIB286` directory is used to contain libraries for 16-bit application development. The `LIB386` directory is used to contain libraries for 32-bit application development.

For 16-bit application development, the Intel x86 processor-dependent libraries are placed under the `\WATCOM\LIB286` directory.

For 32-bit application development, the Intel 386 and upward-compatible processor-dependent libraries are placed under the `\WATCOM\LIB386` directory.

Since Watcom C/C++ also supports several operating systems, including DOS, OS/2, Windows 3.x and Windows NT, system-dependent libraries are grouped under different directories underneath the processor-dependent directories.

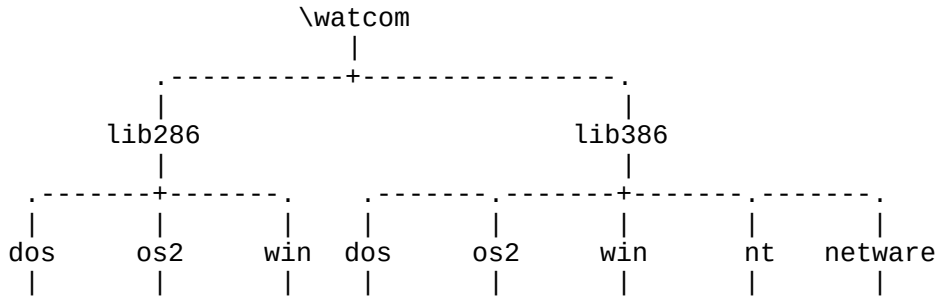
For DOS applications, the system-dependent libraries are placed in `\WATCOM\LIB286\DOS` (16-bit applications) and `\WATCOM\LIB386\DOS` (32-bit applications).

For OS/2 applications, the system-dependent libraries are placed in `\WATCOM\LIB286\OS2` (16-bit applications) and `\WATCOM\LIB386\OS2` (32-bit applications).

For Microsoft Windows applications, the system-dependent libraries are placed in `\WATCOM\LIB286\WIN` (16-bit applications) and `\WATCOM\LIB386\WIN` (32-bit applications).

For Microsoft Windows NT applications, the system-dependent libraries are placed in `\WATCOM\LIB386\NT` (32-bit applications).

For Novell NetWare 386 applications, the system-dependent libraries are placed in `\WATCOM\LIB386\NETWARE` (32-bit applications).



5.2 Watcom C/C++ C Libraries

Due to the many code generation strategies possible in the 80x86 family of processors, a number of versions of the libraries are provided. You must use the libraries which coincide with the particular architecture, operating system, and code generation strategy or model that you have selected. For the type of code generation strategy or model that you intend to use, refer to the description of the "m?" memory model compiler option. The various code models supported by Watcom C/C++ are described in the chapters entitled "16-bit Memory Models" on page 135 and "32-bit Memory Models" on page 217.

We have selected a simple naming convention for the libraries that are provided with Watcom C/C++. Letters are affixed to the file name to indicate the particular strategy with which the modules in the library have been compiled.

16-bit only

S denotes a version of the Watcom C/C++ libraries which have been compiled for the "small" memory model (small code, small data).

M denotes a version of the Watcom C/C++ libraries which have been compiled for the "medium" memory model (big code, small data).

C denotes a version of the Watcom C/C++ libraries which have been compiled for the "compact" memory model (small code, big data).

L denotes a version of the Watcom C/C++ libraries which have been compiled for the "large" memory model (big code, big data).

H denotes a version of the Watcom C/C++ libraries which have been compiled for the "huge" memory model (big code, huge data).

MT denotes a version of the Watcom C/C++ libraries which may be used with OS/2 multi-threaded applications.

DL denotes a version of the Watcom C/C++ libraries which may be used when creating an OS/2 Dynamic Link Library.

32-bit only

3R denotes a version of the Watcom C/C++ libraries that will be used by programs which have been compiled for the "flat/small" memory models using the "3r", "4r" or "5r" option.

3S denotes a version of the Watcom C/C++ libraries that will be used by programs which have been compiled for the "flat/small" memory models using the "3s", "4s" or "5s" option.

The Watcom C/C++ 16-bit libraries are listed below by directory.

Under `\WATCOM\LIB286\DOS`

```
CLIBS.LIB    (DOS small model support)
CLIBM.LIB    (DOS medium model support)
CLIBC.LIB    (DOS compact model support)
CLIBL.LIB    (DOS large model support)
CLIBH.LIB    (DOS huge model support)
GRAPH.LIB    (model independent, DOS graphics support)
```

Under `\WATCOM\LIB286\OS2`

```
CLIBS.LIB    (OS/2 small model support)
CLIBM.LIB    (OS/2 medium model support)
CLIBC.LIB    (OS/2 compact model support)
CLIBL.LIB    (OS/2 large model support)
CLIBH.LIB    (OS/2 huge model support)
CLIBMTL.LIB  (OS/2 multi-thread, large model support)
CLIBDLL.LIB  (OS/2 DLL, large model support)
DOSPMS.LIB   (Phar Lap 286 PM small model support)
DOSPMM.LIB   (Phar Lap 286 PM medium model support)
DOSPMC.LIB   (Phar Lap 286 PM compact model support)
DOSPML.LIB   (Phar Lap 286 PM large model support)
DOSPMH.LIB   (Phar Lap 286 PM huge model support)
```

Under \WATCOM\LIB286\WIN

```
CLIBS.LIB    (Windows small model support)
CLIBM.LIB    (Windows medium model support)
CLIBC.LIB    (Windows compact model support)
CLIBL.LIB    (Windows large model support)
WINDOWS.LIB  (Windows API library)
```

The Watcom C/C++ 32-bit libraries are listed below by directory.

Under \WATCOM\LIB386\DOS

```
CLIB3R.LIB   (flat/small models, "3r", "4r" or "5r" option)
CLIB3S.LIB   (flat/small models, "3s", "4s" or "5s" option)
GRAPH.LIB    (flat/small models, DOS graphics support)
```

The graphics library GRAPH.LIB is independent of the argument passing conventions.

Under \WATCOM\LIB386\OS2

```
CLIB3R.LIB   (flat/small models, "3r", "4r" or "5r" option)
CLIB3S.LIB   (flat/small models, "3s", "4s" or "5s" option)
```

Under \WATCOM\LIB386\WIN

```
CLIB3R.LIB   (flat/small models, "3r", "4r" or "5r" option)
CLIB3S.LIB   (flat/small models, "3s", "4s" or "5s" option)
WIN386.LIB   (32-bit Windows API)
```

Under \WATCOM\LIB386\NT

```
CLIB3R.LIB   (flat/small models, "3r", "4r" or "5r" option)
CLIB3S.LIB   (flat/small models, "3s", "4s" or "5s" option)
```

5.3 Watcom C/C++ Class Libraries

The Watcom C/C++ Class Library routines are described in the ***Watcom C++ Class Library Reference*** manual.

The Watcom C++ 16-bit Class Libraries are listed below.

Under \WATCOM\LIB286

```
    (iostream and string class libraries)
PLIBS.LIB    (small model support)
PLIBM.LIB    (medium model support)
PLIBC.LIB    (compact model support)
PLIBL.LIB    (large model support)
PLIBH.LIB    (huge model support)
PLIBMTL.LIB  (OS/2 multi-thread, large model support)
PLIBDLL.LIB  (OS/2 DLL, large model support)
    (complex class library for "fpc" option)
CPLXS.LIB    (small model support)
CPLXM.LIB    (medium model support)
CPLXC.LIB    (compact model support)
CPLXL.LIB    (large model support)
CPLXH.LIB    (huge model support)
    (complex class library for "fpi..." options)
CPLX7S.LIB   (small model support)
CPLX7M.LIB   (medium model support)
CPLX7C.LIB   (compact model support)
CPLX7L.LIB   (large model support)
CPLX7H.LIB   (huge model support)
```

These libraries are independent of the operating system (except those designated for OS/2). The "7" designates a library compiled with the "7" option.

The Watcom C++ 32-bit Class Libraries are listed below.

Under \WATCOM\LIB386

```
    (iostream and string class libraries)
PLIB3R.LIB   (flat models, "3r", "4r" or "5r" option)
PLIB3S.LIB   (flat models, "3s", "4s" or "5s" option)
PLIBMT3R.LIB (multi-thread library for OS/2 and Windows NT)
PLIBMT3S.LIB (multi-thread library for OS/2 and Windows NT)
    (complex class library for "fpc" option)
CPLX3R.LIB   (flat models, "3r", "4r" or "5r" option)
CPLX3S.LIB   (flat models, "3s", "4s" or "5s" option)
    (complex class library for "fpi..." options)
CPLX73R.LIB  (flat models, "3r", "4r" or "5r" option)
CPLX73S.LIB  (flat models, "3s", "4s" or "5s" option)
```

These libraries are independent of the operating system (except those designated for OS/2 and Windows NT). The "3R" and "3S" suffixes refer to the argument passing convention used. The "7" designates a library compiled with the "7" option.

5.4 Watcom C/C++ MFC Libraries

The Microsoft Foundation Class (MFC) Library routines are described in the on-line help.

The MFC Libraries are provided in source form under `\WATCOM\SRC\MFC`.

5.5 Watcom C/C++ Math Libraries

In general, a Math library is required when floating-point computations are included in the application. The Math libraries are operating-system independent.

For the 286 architecture, the Math libraries are placed under the `\WATCOM\LIB286` directory.

For the 386 architecture, the Math libraries are placed under the `\WATCOM\LIB386` directory.

An 80x87 emulator library, `EMU87.LIB`, is also provided which is both operating-system and architecture dependent.

The following situations indicate that one of the Math libraries should be included when linking the application.

1. When one or more of the functions described in the `MATH.H` header file is referenced, then a Math library must be included.

2. If an application is linked and the message

   ```
   "_fltused_ is an undefined reference"
   ```

   appears, then a Math library must be included.

3. (16-bit only) If an application is linked and the message

   ```
   "__init_87_emulator is an undefined reference"
   ```

   appears, then one of the modules in the application was compiled with one of the "fpi", "fpi87", "fp2", "fp3" or "fp5" options. If the "fpi" option was used, the 80x87 emulator library (`EMU87.LIB`) or the 80x87 fixup library (`NOEMU87.LIB`) should be included when linking the application.

If the "fpi87" option was used, the 80x87 fixup library `NOEMU87.LIB` should be included when linking the application.

The 80x87 emulator is contained in `EMU87.LIB.` Use `NOEMU87.LIB` in place of `EMU87.LIB` when the emulator is not wanted.

4. (32-bit only) If an application is linked and the message

```
"__init_387_emulator is an undefined reference"
```

appears, then one of the modules in the application was compiled with one of the "fpi", "fpi87", "fp2", "fp3" or "fp5" options. If the "fpi" option was used, the 80x87 emulator library (`EMU387.LIB`) should be included when linking the application.

If the "fpi87" option was used, the empty 80x87 emulator library `NOEMU387.LIB` should be included when linking the application.

The 80x87 emulator is contained in `EMU387.LIB.` Use `NOEMU387.LIB` in place of `EMU387.LIB` when the emulator is not wanted.

Normally, the compiler and linker will automatically take care of this. Simply ensure that the **WATCOM** environment variable includes the location of the Watcom C/C++ libraries.

5.6 Watcom C/C++ 80x87 Math Libraries

One of the following Math libraries must be used if any of the modules of your application were compiled with one of the Watcom C/C++ "fpi", "fpi87", "fp2", "fp3" or "fp5" options and your application requires floating-point support for the reasons given above.

16-bit libraries:

```
MATH87S.LIB (small model)
MATH87M.LIB (medium model)
MATH87C.LIB (compact model)
MATH87L.LIB (large model)
MATH87H.LIB (huge model)
NOEMU87.LIB
DOS\EMU87.LIB (DOS dependent)
OS2\EMU87.LIB (OS/2 dependent)
WIN\EMU87.LIB (Windows dependent)
WIN\MATH87C.LIB (Windows dependent)
WIN\MATH87L.LIB (Windows dependent)
```

32-bit libraries:

```
MATH387R.LIB (flat/small models, "3r", "4r" or "5r" option)
MATH387S.LIB (flat/small models, "3s", "4s" or "5s" option)
DOS\EMU387.LIB (DOS dependent)
WIN\EMU387.LIB (Windows dependent)
OS2\EMU387.LIB (OS/2 dependent)
NT\EMU387.LIB (Windows NT dependent)
```

The "fpi" option causes an 80x87 numeric data processor emulator to be linked into your application in addition to any 80x87 math routines that were referenced. This emulator will decode and emulate 80x87 instructions when an 80x87 is not present in the system or if the environment variable **NO87** has been set (this variable is described below).

For 32-bit Watcom Windows-extender applications or 32-bit applications run in Windows 3.1 DOS boxes, you must also include the `WEMU387.386` file in the `[386enh]` section of the `SYSTEM.INI` file.

Example:

```
device=C:\WATCOM\binw\wemu387.386
```

Note that the `WDEBUG.386` file which is installed by the Watcom Installation software contains the emulation support found in the `WEMU387.386` file.

When the "fpi87" option is used exclusively, the emulator is not included. In this case, the application must be run on personal computer systems equipped with the numeric data processor.

5.7 Watcom C/C++ Alternate Math Libraries

One of the following Math libraries must be used if any of the modules of your application were compiled with the Watcom C/C++ "fpc" option and your application requires floating-point support for the reasons given above. The following Math libraries include support for floating-point which is done out-of-line through run-time calls.

16-bit libraries:

```
MATHS.LIB (small model)
MATHM.LIB (medium model)
MATHC.LIB (compact model)
MATHL.LIB (large model)
MATHH.LIB (huge model)
WIN\MATHC.LIB (Windows dependent)
WIN\MATHL.LIB (Windows dependent)
```

32-bit libraries:

```
MATH3R.LIB (flat/small models, "3r", "4r" or "5r" option)
MATH3S.LIB (flat/small models, "3s", "4s" or "5s" option)
```

Applications which are linked with one of these libraries do not require a numeric data processor for floating-point operations. If one is present in the system, it will be used; otherwise floating-point operations are simulated in software. The numeric data processor will not be used if the environment variable **NO87** has been set (this variable is described below).

5.8 The NO87 Environment Variable

If you have a numeric data processor (math coprocessor) in your system but you wish to test a version of your application that will use floating-point emulation ("fpi" option) or simulation ("fpc" option), you can define the **NO87** environment variable.

(16-bit only) The application must be compiled using the "fpc" (floating-point calls) option and linked with the appropriate `MATH?.LIB` library or the "fpi" option (default) and linked with the appropriate `MATH87?.LIB` and `EMU87.LIB` libraries.

(32-bit only) The application must be compiled using the "fpc" (floating-point calls) option and linked with the appropriate `MATH3?.LIB` library or the "fpi" option (default) and linked with the appropriate `MATH387?.LIB` library.

Using the "SET" command, define the environment variable as follows:

```
C>SET NO87=1
```

Now, when you run your application, the 80x87 will be ignored. To undefine the environment variable, enter the command:

```
C>SET NO87=
```

5.9 The Watcom C/C++ Run-time Initialization Routines

Source files are included in the package for the Watcom C/C++ application startup (or initialization) sequence.

(16-bit only) The initialization code directories/files are listed below:

Under \WATCOM\SRC\STARTUP

```
WILDARGV.C    (wild card processing for argv)
8087CW.C      (value loaded into 80x87 control word)
```

Under \WATCOM\SRC\STARTUP\DOS (DOS initialization)

```
CSTRT086.ASM  (startup for 16-bit apps)
DOS16M.ASM    (startup code for Tenberry Software's
DOS/16M)
CMAIN086.C    (final part of initialization sequence)
MDEF.INC      (macros included by assembly code)
```

Under \WATCOM\SRC\STARTUP\WIN (Windows initialization)

```
CSTRTW16.ASM  (startup for 16-bit Windows apps)
LIBENTRY.ASM  (startup for 16-bit Windows DLLs)
MDEF.INC      (macros included by assembly code)
```

Under \WATCOM\SRC\STARTUP\OS2 (OS/2 initialization)

```
CMAIN086.C    (final part of initialization sequence)
MAINO16.C     (middle part of initialization sequence)
CSTRTO16.ASM  (startup for 16-bit OS/2)
EXITWMSG.H    (header file required by MAINO16.C)
WOS2.H        (header file required by MAINO16.C)
INITFINI.H    (header file required by MAINO16.C)
MDEF.INC      (macros included by assembly code)
```

The following is a summary description of the startup files for DOS. The startup files for Windows and OS/2 are similar. The assembler file `CSTRT086.ASM` contains the first part of the initialization code and the remainder is continued in the file `CMAIN086.C`. It is `CMAIN086.C` that calls your main routine `(main)`.

The `DOS16M.ASM` file is a special version of the `CSTRT086.ASM` file which is required when using the Tenberry Software, Inc. DOS/16M 286 DOS extender.

(32-bit only) The initialization code directories/files are listed below:

Under \WATCOM\SRC\STARTUP

```
WILDARGV.C   (wild card processing for argv)
8087CW.C     (value loaded into 80x87 control word)
```

Under \WATCOM\SRC\STARTUP\386

```
CSTRT386.ASM (startup for most DOS Extenders)
CSTRTW32.ASM (startup for 32-bit Windows)
CSTRTX32.ASM (startup for FlashTek DOS Extender)
CMAIN386.C   (final part of initialization sequence)
```

Under \WATCOM\SRC\STARTUP\ADS

```
ADSTART.ASM  (startup for AutoCAD Development System)
```

The assembler files CSTRT*.ASM contain the first part of the initialization code and the remainder is continued in the file CMAIN386.C. It is CMAIN386.C that calls your main routine (main).

The source code is provided for those who wish to customize the initialization sequence for special applications.

The file WILDARGV.C contains an alternate form of "argv" processing in which wild card command line arguments are transformed into lists of matching file names. Wild card arguments are any arguments containing "*" or "?" characters unless the argument is placed within quotes ("). Consider the following example in which we run an application called "TOUCH" with the argument "*.c".

```
C>touch *.c
```

Suppose that the application was linked with the object code for the file WILDARGV.C. Suppose that the files AP1.C, AP2.C and AP3.C are stored in the current directory. The single argument "*.c" is transformed into a list of arguments such that:

```
argc == 4
argv[1] points to "ap1.c"
argv[2] points to "ap2.c"
argv[3] points to "ap3.c"
```

The source file WILDARGV.C must be compiled to produce the object file WILDARGV.OBJ. This file must be specified before the Watcom C/C++ libraries in the linker command file in order to replace the standard "argv" processing.

16-bit Topics

6 16-bit Memory Models

6.1 Introduction

This chapter describes the various 16-bit memory models supported by Watcom C/C++. Each memory model is distinguished by two properties; the code model used to implement function calls and the data model used to reference data.

6.2 16-bit Code Models

There are two code models;

1. the small code model and
2. the big code model.

A small code model is one in which all calls to functions are made with *near calls.* In a near call, the destination address is 16 bits and is relative to the segment value in segment register CS. Hence, in a small code model, all code comprising your program, including library functions, must be less than 64K.

A big code model is one in which all calls to functions are made with *far calls.* In a far call, the destination address is 32 bits (a segment value and an offset relative to the segment value). This model allows the size of the code comprising your program to exceed 64K.

> ***Note:*** If your program contains less than 64K of code, you should use a memory model that employs the small code model. This will result in smaller and faster code since near calls are smaller instructions and are processed faster by the CPU.

6.3 16-bit Data Models

There are three data models;

1. the small data model,
2. the big data model and
3. the huge data model.

A small data model is one in which all references to data are made with *near pointers.* Near pointers are 16 bits; all data references are made relative to the segment value in segment register DS. Hence, in a small data model, all data comprising your program must be less than 64K.

A big data model is one in which all references to data are made with *far pointers.* Far pointers are 32 bits (a segment value and an offset relative to the segment value). This removes the 64K limitation on data size imposed by the small data model. However, when a far pointer is incremented, only the offset is adjusted. Watcom C/C++ assumes that the offset portion of a far pointer will not be incremented beyond 64K. The compiler will assign an object to a new segment if the grouping of data in a segment will cause the object to cross a segment boundary. Implicit in this is the requirement that no individual object exceed 64K bytes. For example, an array containing 40,000 integers does not fit into the big data model. An object such as this should be described as *huge.*

A huge data model is one in which all references to data are made with far pointers. This is similar to the big data model. However, in the huge data model, incrementing a far pointer will adjust the offset *and* the segment if necessary. The limit on the size of an object pointed to by a far pointer imposed by the big data model is removed in the huge data model.

Notes:

1. If your program contains less than 64K of data, you should use the small data model. This will result in smaller and faster code since references using near pointers produce fewer instructions.

2. The huge data model should be used only if needed. The code generated in the huge data model is not very efficient since a run-time routine is called in order to increment far pointers. This increases the size of the code significantly and increases execution time.

6.4 Summary of 16-bit Memory Models

As previously mentioned, a memory model is a combination of a code model and a data model. The following table describes the memory models supported by Watcom C/C++.

Memory Model	Code Model	Data Model	Default Code Pointer	Default Data Pointer
tiny	small	small	near	near
small	small	small	near	near
medium	big	small	far	near
compact	small	big	near	far
large	big	big	far	far
huge	big	huge	far	huge

6.5 Tiny Memory Model

In the tiny memory model, the application's code and data must total less than 64K bytes in size. All code and data are placed in the same segment. Use of the tiny memory model allows the creation of a COM file for the executable program instead of an EXE file. For more information, see the section entitled "Creating a Tiny Memory Model Application" in this chapter.

6.6 Mixed 16-bit Memory Model

A mixed memory model application combines elements from the various code and data models. A mixed memory model application might be characterized as one that uses the ***near,*** ***far,*** or ***huge*** keywords when describing some of its functions or data objects.

For example, a medium memory model application that uses some far pointers to data can be described as a mixed memory model. In an application such as this, most of the data is in a 64K segment (DGROUP) and hence can be referenced with near pointers relative to the segment value in segment register DS. This results in more efficient code being generated and better execution times than one can expect from a big data model. Data objects outside of the DGROUP segment are described with the ***far*** keyword.

6.7 Linking Applications for the Various 16-bit Memory Models

Each memory model requires different run-time and floating-point libraries. Each library assumes a particular memory model and should be linked only with modules that have been compiled with the same memory model. The following table lists the libraries that are to be used to link an application that has been compiled for a particular memory model.

Memory Model	Run-time Library	Floating-Point Calls Library	Floating-Point Library (80x87)
tiny	CLIBS.LIB +CSTART_T.OBJ	MATHS.LIB	MATH87S.LIB +(NO)EMU87.LIB*
small	CLIBS.LIB	MATHS.LIB	MATH87S.LIB +(NO)EMU87.LIB*
medium	CLIBM.LIB	MATHM.LIB	MATH87M.LIB +(NO)EMU87.LIB*
compact	CLIBC.LIB	MATHC.LIB	MATH87C.LIB +(NO)EMU87.LIB*
large	CLIBL.LIB	MATHL.LIB	MATH87L.LIB +(NO)EMU87.LIB*
huge	CLIBH.LIB	MATHH.LIB	MATH87H.LIB +(NO)EMU87.LIB*

* One of `EMU87.LIB` or `NOEMU87.LIB` will be used with the 80x87 math libraries depending on the use of the "fpi" (include emulation) or "fpi87" (do not include emulation) options.

6.8 Creating a Tiny Memory Model Application

Tiny memory model programs are created by compiling all modules with the small memory model option and linking in the special initialization file "CSTART_T.OBJ". This file is found in the Watcom C/C++ `LIB286\DOS` directory. It must be the first object file specified when linking the program.

The following sequence will create the executable file "MYPROG.COM" from the file "MYPROG.C":

Example:

```
C>wcc myprog /ms
C>wlink system com file myprog
```

Most of the details of linking a "COM" program are handled by the "SYSTEM COM" directive (see the `WLSYSTEM.LNK` file for details). When linking a "COM" program, the message "Stack segment not found" is issued. This message may be ignored.

6.9 Memory Layout

The following describes the segment ordering of an application linked by the Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all segments not belonging to group "DGROUP" with class "CODE"
2. all other segments not belonging to group "DGROUP"
3. all segments belonging to group "DGROUP" with class "BEGDATA"
4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"
5. all segments belonging to group "DGROUP" with class "BSS"
6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with Watcom run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Watcom C/C++.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<module>_TEXT" where <module> is the file name of the source file.

2. The "FAR_DATA" class consists of the following:

   (a) data objects whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (b) data objects defined using the "FAR" or "HUGE" keyword,

   (c) literals whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (d) literals defined using the "FAR" or "HUGE" keyword.

You can override the default naming convention used by Watcom C/C++ to name segments.

1. The Watcom C/C++ "nm" option can be used to change the name of the module. This, in turn, changes the name of the code segment when compiling for a big code model.
2. The Watcom C/C++ "nt" option can be used to specify the name of the code segment regardless of the code model used.

7 16-bit Assembly Language Considerations

7.1 Introduction

This chapter will deal with the following topics.

1. The data representation of the basic types supported by Watcom C/C++.
2. The memory layout of a Watcom C/C++ program.
3. The method for passing arguments and returning values.
4. The two methods for passing floating-point arguments and returning floating-point values.

   One method is used when one of the Watcom C/C++ "fpi" or "fpi87" options is specified for the generation of in-line 80x87 instructions. When the "fpi" option is specified, an 80x87 emulator is included from a math library if the application includes floating-point operations. When the "fpi87" option is used exclusively, the 80x87 emulator will not be included.

   The other method is used when the Watcom C/C++ "fpc" option is specified. In this case, the compiler generates calls to floating-point support routines in the alternate math libraries.

An understanding of the Intel 80x86 architecture is assumed.

7.2 Data Representation

This section describes the internal or machine representation of the basic types supported by Watcom C/C++.

7.2.1 Type "char"

An item of type "char" occupies 1 byte of storage. Its value is in the following range.

```
0 <= n <= 255
```

Note that "char" is, by default, unsigned. The Watcom C/C++ compiler option "j" can be used to change the default from unsigned to signed. If "char" is signed, an item of type "char" is in the following range.

```
-128 <= n <= 127
```

You can force an item of type "char" to be unsigned or signed regardless of the default by defining them to be of type "unsigned char" or "signed char" respectively.

7.2.2 Type "short int"

An item of type "short int" occupies 2 bytes of storage. Its value is in the following range.

```
-32768 <= n <= 32767
```

Note that "short int" is signed and hence "short int" and "signed short int" are equivalent. If an item of type "short int" is to be unsigned, it must be defined as "unsigned short int". In this case, its value is in the following range.

```
0 <= n <= 65535
```

7.2.3 Type "long int"

An item of type "long int" occupies 4 bytes of storage. Its value is in the following range.

```
-2147483648 <= n <= 2147483647
```

Note that "long int" is signed and hence "long int" and "signed long int" are equivalent. If an item of type "long int" is to be unsigned, it must be defined as "unsigned long int". In this case, its value is in the following range.

```
0 <= n <= 4294967295
```

7.2.4 Type "int"

An item of type "int" occupies 2 bytes of storage. Its value is in the following range.

```
-32768 <= n <= 32767
```

Note that "int" is signed and hence "int" and "signed int" are equivalent. If an item of type "int" is to be unsigned, it must be defined as "unsigned int". In this case its value is in the following range.

```
0 <= n <= 65535
```

If you are generating code that executes in 16-bit mode, "short int" and "int" are equivalent, "unsigned short int" and "unsigned int" are equivalent, and "signed short int" and "signed int" are equivalent. This may not be the case in other environments where "int" and "long int" are 4 bytes.

7.2.5 Type "float"

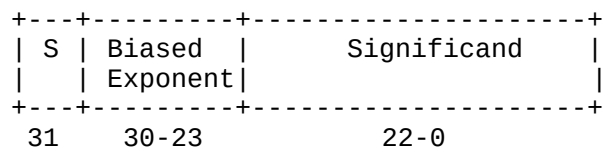
A datum of type "float" is an approximate representation of a real number. Each datum of type "float" occupies 4 bytes. If `m` is the magnitude of `x` (an item of type "float") then `x` can be approximated if

$$2^{-126} <= m < 2^{128}$$

or in more approximate terms if

```
1.175494e-38 <= m <= 3.402823e38
```

Data of type "float" are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.



Notes

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 127 (i.e., exponent value 1 represents 2^{-126}; exponent value 127 represents 2^{0}; exponent value 254 represents 2^{127}; etc.). The exponent field is 8 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 23 bits long.

Zero A real zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

7.2.6 Type "double"

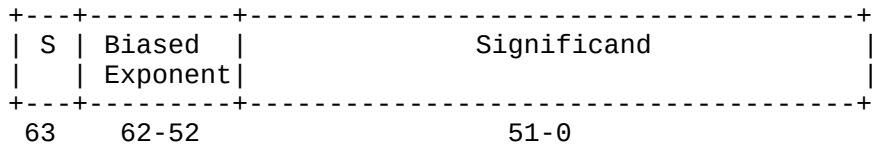
A datum of type "double" is an approximate representation of a real number. The precision of a datum of type "double" is greater than or equal to one of type "float". Each datum of type "double" occupies 8 bytes. If `m` is the magnitude of `x` (an item of type "double") then `x` can be approximated if

$$2^{-1022} <= m < 2^{1024}$$

or in more approximate terms if

```
2.2250738585072e-308 <= m <= 1.79769313486232e308
```

Data of type "double" are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.



Notes:

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 1023 (i.e., exponent value 1 represents 2^{-1022}; exponent value 1023 represents 2^{0}; exponent value 2046 represents 2^{1023}; etc.). The exponent field is 11 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 52 bits long.

Zero A double precision zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

7.3 Memory Layout

The following describes the segment ordering of an application linked by the Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all segments not belonging to group "DGROUP" with class "CODE"
2. all other segments not belonging to group "DGROUP"
3. all segments belonging to group "DGROUP" with class "BEGDATA"

4. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

5. all segments belonging to group "DGROUP" with class "BSS"

6. all segments belonging to group "DGROUP" with class "STACK"

A special segment belonging to class "BEGDATA" is defined when linking with Watcom run-time libraries. This segment is initialized with the hexadecimal byte pattern "01" and is the first segment in group "DGROUP" so that storing data at location 0 can be detected.

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Watcom C/C++.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<module>_TEXT" where <module> is the file name of the source file.

2. The "FAR_DATA" class consists of the following:

   (a) data objects whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (b) data objects defined using the "FAR" or "HUGE" keyword,

   (c) literals whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (d) literals defined using the "FAR" or "HUGE" keyword.

You can override the default naming convention used by Watcom C/C++ to name segments.

1. The Watcom C/C++ "nm" option can be used to change the name of the module. This, in turn, changes the name of the code segment when compiling for a big code model.

2. The Watcom C/C++ "nt" option can be used to specify the name of the code segment regardless of the code model used.

7.4 Calling Conventions for Non-80x87 Applications

The following sections describe the calling convention used when compiling with the "fpc" compiler option.

7.4.1 Passing Arguments Using Register-Based Calling Conventions

How arguments are passed to a function with register-based calling conventions is determined by the size (in bytes) of the argument and where in the argument list the argument appears. Depending on the size, arguments are either passed in registers or on the stack. Arguments such as structures are almost always passed on the stack since they are generally too large to fit in registers. Since arguments are processed from left to right, the first few arguments are likely to be passed in registers (if they can fit) and, if the argument list contains many arguments, the last few arguments are likely to be passed on the stack.

The registers used to pass arguments to a function are AX, BX, CX and DX. The following algorithm describes how arguments are passed to functions.

Initially, we have the following registers available for passing arguments: AX, DX, BX and CX. Note that registers are selected from this list in the order they appear. That is, the first register selected is AX and the last is CX. For each argument *Ai*, starting with the left most argument, perform the following steps.

1. If the size of *Ai* is 1 byte, convert it to 2 bytes and proceed to the next step. If *Ai* is of type "unsigned char", it is converted to an "unsigned int". If *Ai* is of type "signed char", it is converted to a "signed int". If *Ai* is a 1-byte structure, the padding is determined by the compiler.

2. If an argument has already been assigned a position on the stack, *Ai* will also be assigned a position on the stack. Otherwise, proceed to the next step.

3. If the size of *Ai* is 2 bytes, select a register from the list of available registers. If a register is available, *Ai* is assigned that register. The register is then removed from the list of available registers. If no registers are available, *Ai* will be assigned a position on the stack.

4. If the size of *Ai* is 4 bytes, select a register pair from the following list of combinations: [DX AX] or [CX BX]. The first available register pair is assigned

to *Ai* and removed from the list of available pairs. The high-order 16 bits of the argument are assigned to the first register in the pair; the low-order 16 bits are assigned to the second register in the pair. If none of the above register pairs is available, *Ai* will be assigned a position on the stack.

5. If the type of *Ai* is "double" or "float" (in the absence of a function prototype), select [AX BX CX DX] from the list of available registers. All four registers are removed from the list of available registers. The high-order 16 bits of the argument are assigned to the first register and the low-order 16 bits are assigned to the fourth register. If any of the four registers is not available, *Ai* will be assigned a position on the stack.

6. All other arguments will be assigned a position on the stack.

Notes:

1. Arguments that are assigned a position on the stack are padded to a multiple of 2 bytes. That is, if a 3-byte structure is assigned a position on the stack, 4 bytes will be pushed on the stack.

2. Arguments that are assigned a position on the stack are pushed onto the stack starting with the rightmost argument.

7.4.2 Sizes of Predefined Types

The following table lists the predefined types, their size as returned by the "sizeof" function, the size of an argument of that type and the registers used to pass that argument if it was the only argument in the argument list.

Basic Type	*"sizeof"*	*Argument Size*	*Registers Used*
char	1	2	[AX]
short int	2	2	[AX]
int	2	2	[AX]
long int	4	4	[DX AX]
float	4	8	[AX BX CX DX]
double	8	8	[AX BX CX DX]
near pointer	2	2	[AX]
far pointer	4	4	[DX AX]
huge pointer	4	4	[DX AX]

Note that the size of the argument listed in the table assumes that no function prototypes are specified. Function prototypes affect the way arguments are passed. This will be discussed in the section entitled "Effect of Function Prototypes on Arguments".

Notes:

1. Provided no function prototypes exist, an argument will be converted to a default type as described in the following table.

Argument Type	***Passed As***
char	unsigned int
signed char	signed int
unsigned char	unsigned int
float	double

7.4.3 Size of Enumerated Types

The integral type of an enumerated type is determined by the values of the enumeration constants. In strict ANSI C mode, all enumerated constants are of type `int.` In the extensions mode, the compiler will use the smallest integral type possible (excluding `long` ints) that can represent all values of the enumerated type. For instance, if the minimum and maximum values of the enumeration constants are in the range −128 and 127, the enumerated type will be equivalent to a `signed char` (size = 1 byte). All references to enumerated constants in the previous instance will have type `signed char.` An enumerated constant is always promoted to an `int` when passed as an argument.

7.4.4 Effect of Function Prototypes on Arguments

Function prototypes define the types of the formal parameters of a function. Their appearance affects the way in which arguments are passed. An argument will be converted to the type of the corresponding formal parameter in the function prototype. Consider the following example.

```
void prototype( float x, int i );

void main()
{
  float x;
  int   i;

  x = 3.14;
  i = 314;
  prototype( x, i );
  rtn( x, i );
}
```

The function prototype for `prototype` specifies that the first argument is to be passed as a "float" and the second argument is to be passed as an "int". This results in the first argument being passed in registers DX and AX and the second argument being passed in register BX.

If no function prototype is given, as is the case for the function `rtn,` the first argument will be passed as a "double" and the second argument would be passed as an "int". This results in the first argument being passed in registers AX, BX, CX and DX and the second argument being passed on the stack.

Note that even though both `prototype` and `rtn` were called with identical argument lists, the way in which the arguments were passed was completely different simply because a function prototype for `prototype` was specified. Function prototyping is an excellent way to guarantee that arguments will be passed as expected to your assembly language function.

7.4.5 Interfacing to Assembly Language Functions

Consider the following example.

Example:

```
void main()
{
    long int x;
    int      i;
    long int y;

    x = 7;
    i = 77;
    y = 777;
    myrtn( x, i, y );
}
```

`myrtn` is an assembly language function that requires three arguments. The first argument is of type "long int", the second argument is of type "int" and the third argument is again of type "long int". Using the rules for register-based calling conventions, these arguments will be passed to `myrtn` in the following way:

1. The first argument will be passed in registers DX and AX leaving BX and CX as available registers for other arguments.

2. The second argument will be passed in register BX leaving CX as an available register for other arguments.

3. The third argument will not fit in register CX (its size is 4 bytes) and hence will be pushed on the stack.

Let us look at the stack upon entry to `myrtn.`

Small Code Model

```
Offset
        +-----------------+
  0     | return address | <- SP points here
        +-----------------+
  2     | argument #3    |
        |                |
        +-----------------+
  6     |                |
```

Big Code Model

```
Offset
        +-----------------+
  0     | return address | <- SP points here
        |                |
        +-----------------+
  4     | argument #3    |
        |                |
        +-----------------+
  8     |                |
```

Notes:

1. The return address is the top element on the stack. In a small code model, the return address is 1 word (16 bits)

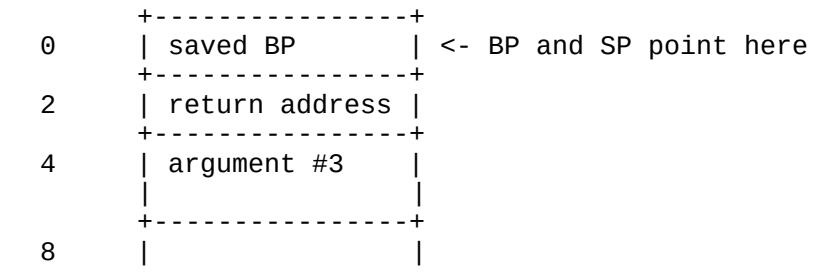
Register SP cannot be used as a base register to address the third argument on the stack. Register BP is normally used to address arguments on the stack. Upon entry to the function,

register BP is set to point to the stack but before doing so we must save its contents. The following two instructions achieve this.

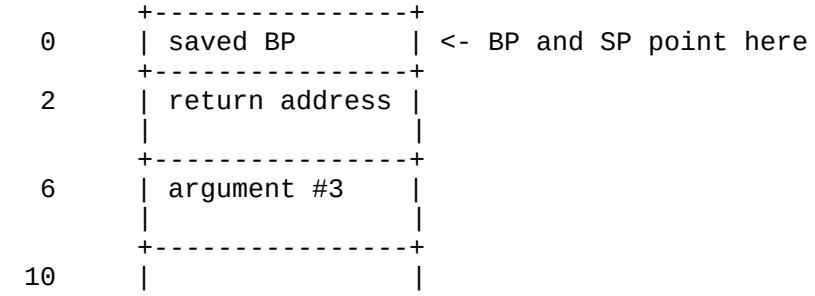
```
push    BP          ; save current value of BP
mov     BP,SP       ; get access to arguments
```

After executing these instructions, the stack looks like this.

Small Code Model



Big Code Model



As the above diagrams show, the third argument is at offset 4 from register BP in a small code model and offset 6 in a big code model.

Upon exit from `myrtn`, we must restore the value of BP. The following two instructions achieve this.

```
mov     SP,BP       ; restore stack pointer
pop     BP          ; restore BP
```

The following is a sample assembly language function which implements `myrtn`.

Small Memory Model (small code, small data)

```
DGROUP     group     _DATA, _BSS
_TEXT      segment byte public 'CODE'
           assume    CS:_TEXT
           assume    DS:DGROUP
           public    myrtn_
myrtn_     proc      near
           push      BP          ; save BP
           mov       BP,SP       ; get access to arguments
;
; body of function
;
           mov       SP,BP       ; restore SP
           pop       BP          ; restore BP
           ret       4           ; return and pop last arg
myrtn_     endp
_TEXT      ends
```

Large Memory Model (big code, big data)

```
DGROUP     group     _DATA, _BSS
MYRTN_TEXT segment byte public 'CODE'
           assume    CS:MYRTN_TEXT
           public    myrtn_
myrtn_     proc      far
           push      BP          ; save BP
           mov       BP,SP       ; get access to arguments
;
; body of function
;
           mov       SP,BP       ; restore SP
           pop       BP          ; restore BP
           ret       4           ; return and pop last arg
myrtn_     endp
MYRTN_TEXT ends
```

Notes:

1. Global function names must be followed with an underscore. Global variable names must be preceded with an underscore.

2. All used 80x86 registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to saved and restored if you are compiling your application with the "r" option.

3. The direction flag must be clear before returning to the caller.

4. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.

5. In a small data model, segment register DS contains the segment address of the group "DGROUP". This is not the case in a big data model.

6. When writing assembly language functions for the small code model, you must declare them as "near". If you wish to write assembly language functions for the big code model, you must declare them as "far".

7. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

8. If any of the arguments was pushed onto the stack, the called routine must pop those arguments off the stack in the "ret" instruction.

7.4.6 Functions with Variable Number of Arguments

A function prototype with a parameter list that ends with ",..." has a variable number of arguments. In this case, all arguments are passed on the stack. Since no prototyping information exists for arguments represented by ",...", those arguments are passed as described in the section "Passing Arguments".

7.4.7 Returning Values from Functions

The way in which function values are returned depends on the size of the return value. The following examples describe how function values are to be returned. They are coded for a small code model.

1. 1-byte values are to be returned in register AL.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret1_
Ret1_   proc    near   ; char Ret1()
        mov     AL,'G'
        ret
Ret1_   endp
_TEXT   ends
        end
```

2. 2-byte values are to be returned in register AX.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret2_
Ret2_   proc    near   ; short int Ret2()
        mov     AX,77
        ret
Ret2_   endp
_TEXT   ends
        end
```

3. 4-byte values are to be returned in registers DX and AX with the most significant word in register DX.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret4_
Ret4_   proc    near   ; long int Ret4()
        mov     AX,word ptr CS:Val4+0
        mov     DX,word ptr CS:Val4+2
        ret
Val4    dd      7777777
Ret4_   endp
_TEXT   ends
        end
```

4. 8-byte values, except structures, are to be returned in registers AX, BX, CX and DX with the most significant word in register AX.

Example:

```
        .8087
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret8_
Ret8_   proc    near   ; double Ret8()
        mov     DX,word ptr CS:Val8+0
        mov     CX,word ptr CS:Val8+2
        mov     BX,word ptr CS:Val8+4
        mov     AX,word ptr CS:Val8+6
        ret
Val8:   dq      7.7
Ret8_   endp
_TEXT   ends
        end
```

The ".8087" pseudo-op must be specified so that all floating-point constants are generated in 8087 format. When using the "fpc" (floating-point calls) option, "float" and "double" are returned in registers. See section "Returning Values in 80x87-based Applications" when using the "fpi" or "fpi87" options.

5. Otherwise, the caller allocates space on the stack for the return value and sets register SI to point to this area. In a big data model, register SI contains an offset relative to the segment value in segment register SS.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  RetX_
;
; struct int_values {
;     int value1, value2, value3, value4, value5;
;                   };
;
RetX_   proc    near ; struct int_values RetX()
        mov     word ptr SS:0[SI],71
        mov     word ptr SS:4[SI],72
        mov     word ptr SS:8[SI],73
        mov     word ptr SS:12[SI],74
        mov     word ptr SS:16[SI],75
        ret
RetX_   endp
_TEXT   ends
        end
```

When returning values on the stack, remember to use a segment override to the stack segment (SS).

The following is an example of a Watcom C/C++ program calling the above assembly language subprograms.

```
#include <stdio.h>

struct int_values {
    int value1;
    int value2;
    int value3;
    int value4;
    int value5;
};

extern  char              Ret1(void);
extern  short int         Ret2(void);
extern  long int          Ret4(void);
extern  double            Ret8(void);
extern  struct int_values RetX(void);

void main()
{
    struct int_values x;

    printf( "Ret1 = %c\n", Ret1() );
    printf( "Ret2 = %d\n", Ret2() );
    printf( "Ret4 = %ld\n", Ret4() );
    printf( "Ret8 = %f\n", Ret8() );
    x = RetX();
    printf( "RetX1 = %d\n", x.value1 );
    printf( "RetX2 = %d\n", x.value2 );
    printf( "RetX3 = %d\n", x.value3 );
    printf( "RetX4 = %d\n", x.value4 );
    printf( "RetX5 = %d\n", x.value5 );
}
```

The above function should be compiled for a small code model (use the "ms" or "mc" compiler option).

7.5 Calling Conventions for 80x87-based Applications

When a source file is compiled by Watcom C/C++ with one of the "fpi" or "fpi87" options, all floating-point arguments are passed on the 80x86 stack. The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, it is assigned a position on the 80x86 stack.

7.5.1 Passing Values in 80x87-based Applications

Consider the following example.

Example:

```
extern  void    myrtn(int,float,double,long int);

void main()
{
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77
    j = 77;
    myrtn( i, x, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument is of type "int" (2 bytes), the second argument is of type "float" (4 bytes), the third argument is of type "double" (8 bytes) and the fourth argument is of type "long int" (4 bytes). These arguments will be passed to `myrtn` in the following way:

1. The first argument will be passed in register AX leaving BX, CX and DX as available registers for other arguments.

2. The second argument will be passed on the 80x86 stack since it is a floating-point argument.

3. The third argument will also be passed on the 80x86 stack since it is a floating-point argument.

4. The fourth argument will be passed on the 80x86 stack since a previous argument has been assigned a position on the 80x86 stack.

Remember, arguments are pushed on the stack from right to left. That is, the rightmost argument is pushed first.

Any assembly language function must obey the following rule.

1. All arguments passed on the stack must be removed by the called function.

The following is a sample assembly language function which implements `myrtn`.

Example:

```
        .8087
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  myrtn_
myrtn_  proc    near
;
; body of function
;
        ret 16          ; return and pop arguments
myrtn_  endp
_TEXT   ends
        end
```

Notes:

1. Function names must be followed by an underscore.

2. All used 80x86 registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to saved and restored if you are compiling your application with the "r" option. In this example, AX does not have to be saved as it was used to pass the first argument. Floating-point registers can be modified without saving their contents.

3. The direction flag must be clear before returning to the caller.

4. This function has been written for a small code model. Any segment containing executable code must belong to the class "CODE" and the segment "_TEXT". On entry, CS contains the segment address of the segment "_TEXT". The above restrictions do not apply in a big code memory model.

5. When writing assembly language functions for a small code model, you must declare them as "near". If you wish to write assembly language functions for a big code model, you must declare them as "far".

7.5.2 Returning Values in 80x87-based Applications

Floating-point values are returned in ST(0) when using the "fpi" or "fpi87" options. All other values are returned in the manner described earlier in this chapter.

8 16-bit Pragmas

8.1 Introduction

A pragma is a compiler directive that provides the following capabilities.

- Pragmas allow you to specify certain compiler options.
- Pragmas can be used to direct the Watcom C/C++ code generator to emit specialized sequences of code for calling functions which use argument passing and value return techniques that differ from the default used by Watcom C/C++.
- Pragmas can be used to describe attributes of functions (such as side effects) that are not possible at the C/C++ language level. The code generator can use this information to generate more efficient code.
- Any sequence of in-line machine language instructions, including DOS and BIOS function calls, can be generated in the object code.

Pragmas are specified in the source file using the ***pragma*** directive. The following notation is used to describe the syntax of pragmas.

keywords A `keyword` is shown in a mono-spaced courier font.

program-item A ***program-item*** is shown in a roman bold-italics font. A ***program-item*** is a symbol name or numeric value supplied by the programmer.

punctuation A `punctuation character` shown in a mono-spaced courier font must be entered as is.

A ***punctuation character*** shown in a roman bold-italics font is used to describe syntax. The following syntactical notation is used.

[abc] The item *abc* is optional.

{abc} The item *abc* may be repeated zero or more times.

a|b|c One of *a*, *b* or *c* may be specified.

a ::= b The item *a* is defined in terms of *b*.

(a) Item *a* is evaluated first.

The following classes of pragmas are supported.

- pragmas that specify options
- pragmas that specify default libraries
- pragmas that describe the way structures are stored in memory
- pragmas that provide auxiliary information used for code generation

8.2 Using Pragmas to Specify Options

Currently, the following options can be specified with pragmas:

unreferenced The "unreferenced" option controls the way Watcom C/C++ handles unused symbols. For example,

```
#pragma on (unreferenced);
```

will cause Watcom C/C++ to issue warning messages for all unused symbols. This is the default. Specifying

```
#pragma off (unreferenced);
```

will cause Watcom C/C++ to ignore unused symbols. Note that if the warning level is not high enough, warning messages for unused symbols will not be issued even if "unreferenced" was specified.

check_stack The "check_stack" option controls the way stack overflows are to be handled. For example,

```
#pragma on (check_stack);
```

will cause stack overflows to be detected and

```
#pragma off (check_stack);
```

will cause stack overflows to be ignored. When "check_stack" is on, Watcom C/C++ will generate a run-time call to a stack-checking routine at the start of every routine compiled. This run-time routine will issue an error if a stack overflow occurs when invoking the routine. The default is to check for stack overflows. Stack overflow checking is particularly useful when functions are invoked recursively. Note that if the stack overflows and stack checking has been suppressed, unpredictable results can occur.

If a stack overflow does occur during execution and you are sure that your program is not in error (i.e. it is not unnecessarily recursing), you must increase the stack size. This is done by linking your application again and specifying the "STACK" option to the Watcom Linker with a larger stack size.

It is also possible to specify more than one option in a pragma as illustrated by the following example.

```
#pragma on (check_stack unreferenced);
```

reuse_duplicate_strings (C only) (C Only) The "reuse_duplicate_strings" option controls the way Watcom C handles identical strings in an expression. For example,

```
#pragma on (reuse_duplicate_strings);
```

will cause Watcom C to reuse identical strings in an expression. This is the default. Specifying

```
#pragma off (reuse_duplicate_strings);
```

will cause Watcom C to generate additional copies of the identical string. The following example shows where this may be of importance to the way the application behaves.

Example:

```
#include <stdio.h>

#pragma off (reuse_duplicate_strings)


void poke( char *, char * );

void main()
  {
    poke( "Hello world\n", "Hello world\n" );
  }

void poke( char *x, char *y )
  {
    x[3] = 'X';
    printf( x );
    y[4] = 'Y';
    printf( y );
  }
/*
Default output:
HelXo world
HelXY world
*/
```

8.3 Using Pragmas to Specify Default Libraries

Default libraries are specified in special object module records. Library names are extracted from these special records by the Watcom Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched.

By default, that is if no library pragma is specified, the Watcom C/C++ compiler generates, in the object file defining the main program, default libraries corresponding to the memory model and floating-point model used to compile the file. For example, if you have compiled the source file containing the main program for the medium memory model and the floating-point calls floating-point model, the libraries "clibm" and "mathm" will be placed in the object file.

If you wish to add your own default libraries to this list, you can do so with a library pragma. Consider the following example.

```
#pragma library (mylib);
```

The name "mylib" will be added to the list of default libraries specified in the object file.

If the library specification contains characters such as '\', ':' or ',' (i.e., any character not allowed in a C identifier), you must enclose it in double quotes as in the following example.

```
#pragma library ("\watcom\lib286\dos\graph.lib");
#pragma library ("\watcom\lib386\dos\graph.lib");
```

If you wish to specify more than one library in a library pragma you must separate them with spaces as in the following example.

```
#pragma library (mylib "\watcom\lib286\dos\graph.lib");
#pragma library (mylib "\watcom\lib386\dos\graph.lib");
```

8.4 The ALLOC_TEXT Pragma (C Only)

The "alloc_text" pragma can be used to specify the name of the text segment into which the generated code for a function, or a list of functions, is to be placed. The following describes the form of the "alloc_text" pragma.

```
#pragma alloc_text ( seg_name, fn {, fn} ) [;]
```

where ***description:***

seg_name is the name of the text segment.

fn is the name of a function.

Consider the following example.

```
extern int fn1(int);
extern int fn2(void);
#pragma alloc_text ( my_text, fn1, fn2 );
```

The code for the functions `fn1` and `fn2` will be placed in the segment `my_text`. Note: function prototypes for the named functions must exist prior to the "alloc_text" pragma.

8.5 The CODE_SEG Pragma

The "code_seg" pragma can be used to specify the name of the text segment into which the generated code for functions is to be placed. The following describes the form of the "code_seg" pragma.

```
#pragma code_seg ( seg_name [, class_name] ) [;]
```

where ***description:***

seg_name is the name of the text segment enclosed in quotes. Also, `seg_name` may be a macro as in:

```
#define seg_name "MY_CODE_SEG"
#pragma code_seg ( seg_name );
```

class_name is the optional class name of the text segment enclosed in quotes. Also, `class_name` may be a macro as in:

```
#define class_name "MY_CLASS"
#pragma code_seg ( "MY_CODE_SEG", class_name );
```

Consider the following example.

```
#pragma code_seg ( "my_text" );

int incr( int i )
{
    return( i + 1 );
}

int decr( int i )
{
    return( i - 1 );
}
```

The code for the functions `incr` and `decr` will be placed in the segment `my_text.`

To return to the default segment, do not specify any string between the opening and closing parenthesis.

```
#pragma code_seg ();
```

8.6 The COMMENT Pragma

The "comment" pragma can be used to place a comment record in an object file or executable file. The following describes the form of the "comment" pragma.

```
#pragma comment ( comment_type [, "comment_string"] ) [;]
```

where description:

comment_type specifies the type of comment record. The allowable comment types are:

lib	Default libraries are specified in special object module records. Library names are extracted from these special records by the Watcom Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched. The "lib" form of this pragma offers the same features as the "library" pragma. See the section entitled "Using Pragmas to Specify Default Libraries" on page 164 for more information.

"comment_string" is an optional string literal that provides additional information for some comment types.

Consider the following example.

```
#pragma comment ( lib, "mylib" );
```

8.7 The DATA_SEG Pragma

The "data_seg" pragma can be used to specify the name of the segment into which data is to be placed. The following describes the form of the "data_seg" pragma.

```
#pragma data_seg ( seg_name [, class_name] ) [;]
```

where ***description:***

seg_name is the name of the data segment enclosed in quotes. Also, `seg_name` may be a macro as in:

```
#define seg_name "MY_DATA_SEG"
#pragma data_seg ( seg_name );
```

class_name is the optional class name of the data segment enclosed in quotes. Also, `class_name` may be a macro as in:

```
#define class_name "MY_CLASS"
#pragma data_seg ( "MY_DATA_SEG", class_name );
```

Consider the following example.

```
#pragma data_seg ( "my_data" );

static int i;
static int j;
```

The data for `i` and `j` will be placed in the segment `my_data`.

To return to the default segment, do not specify any string between the opening and closing parenthesis.

```
#pragma data_seg ();
```

8.8 The DISABLE_MESSAGE Pragma (C Only)

The "disable_message" pragma disables the issuance of specified diagnostic messages. The form of the "disable_message" pragma is as follows.

```
#pragma disable_message ( msg_num {, msg_num} ) [;]
```

where ***description:***

msg_num is the number of the diagnostic message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Watcom C Diagnostic Messages" on page 363. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

See also the description of "The ENABLE_MESSAGE Pragma (C Only)".

8.9 The DUMP_OBJECT_MODEL Pragma (C++ Only)

The "dump_object_model" pragma causes the C++ compiler to print information about the object model for an indicated class or an enumeration name to the diagnostics file. For class names, this information includes the offsets and sizes of fields within the class and within base classes. For enumeration names, this information consists of a list of all the enumeration constants with their values.

The general form of the "dump_object_model" pragma is as follows.

```
#pragma dump_object_model class [;]
#pragma dump_object_model enumeration [;]
class ::= a defined C++ class free of errors
enumeration ::= a defined C++ enumeration name
```

This pragma is designed to be used for information purposes only.

8.10 The ENABLE_MESSAGE Pragma (C Only)

The "enable_message" pragma re-enables the issuance of specified diagnostic messages that have been previously disabled. The form of the "enable_message" pragma is as follows.

```
#pragma enable_message ( msg_num {, msg_num} ) [;]
```

where ***description:***

msg_num is the number of the diagnostic message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Watcom C Diagnostic Messages" on page 363. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

See also the description of "The DISABLE_MESSAGE Pragma (C Only)" on page 168.

8.11 The ENUM Pragma

The "enum" pragma affects the underlying storage-definition for subsequent *enum* declarations. The forms of the "enum" pragma are as follows.

```
#pragma enum int [;]
#pragma enum minimum [;]
#pragma enum original [;]
#pragma enum pop [;]
```

where ***description:***

int Make *int* the underlying storage definition (same as the "ei" compiler option).

minimum Minimize the underlying storage definition (same as not specifying the "ei" compiler option).

original Reset back to the original compiler option setting (i.e., what was or was not specified on the command line).

pop Restore the previous setting.

The first three forms all push the previous setting before establishing the new setting.

8.12 The ERROR Pragma

The "error" pragma can be used to issue an error message with the specified text. The following describes the form of the "error" pragma.

```
#pragma error "error text" [;]
```

where ***description:***

"error text" is the text of the message that you wish to display.

You should use the ANSI ***#error*** directive rather than this pragma. This pragma is provided for compatibility with legacy code. The following is an example.

```
#if defined(__386__)
    ...
#elseif defined(__86__)
    ...
#else
#pragma error ( "neither __386__ or __86__ defined" );
#endif
```

8.13 The EXTREF Pragma

The "extref" pragma is used to generate a reference to an external function or data item. The form of the "extref" pragma is as follows.

```
#pragma extref name [;]
```

where ***description:***

name is the name of an external function or data item. It must be declared to be an external function or data item before the pragma is encountered. In C++, when *name* is a function, it must not be overloaded.

This pragma causes an external reference for the function or data item to be emitted into the object file even if that function or data item is not referenced in the module. The external

reference will cause the linker to include the module containing that name in the linked program or DLL.

This is useful for debugging since you can cause debugging routines (callable from within debugger) to be included into a program or DLL to be debugged.

In C++, you can also force constructors and/or destructors to be called for a data item without necessarily referencing the data item anywhere in your code.

8.14 The FUNCTION Pragma

Certain functions, such as those listed in the description of the "oi" and "om" options, have intrinsic forms. These functions are special functions that are recognized by the compiler and processed in a special way. For example, the compiler may choose to generate in-line code for the function. The intrinsic attribute for these special functions is set by specifying the "oi" or "om" option or using an "intrinsic" pragma. The "function" pragma can be used to remove the intrinsic attribute for a specified list of functions.

The following describes the form of the "function" pragma.

```
#pragma function ( fn {, fn} ) [;]
```

where ***description:***

fn is the name of a function.

Suppose the following source code was compiled using the "om" option so that when one of the special math functions is referenced, the intrinsic form will be used. In our example, we have referenced the function `sin` which does have an intrinsic form. By specifying `sin` in a "function" pragma, the intrinsic attribute will be removed, causing the function `sin` to be treated as a regular user-defined function.

```
#include <math.h>
#pragma function( sin );

double test( double x )
{
    return( sin( x ) );
}
```

8.15 Setting Priority of Static Data Initialization (C++ Only)

The "initialize" pragma sets the priority for initialization of static data in the file. This priority only applies to initialization of static data that requires the execution of code. For example, the initialization of a class that contains a constructor requires the execution of the constructor. Note that if the sequence in which initialization of static data in your program takes place has no dependencies, the "initialize" pragma need not be used.

The general form of the "initialize" pragma is as follows.

```
#pragma initialize [before | after] priority [;]

priority ::= n | library | program
```

where ***description:***

n is a number representing the priority and must be in the range 0-255. The larger the priority, the later the point at which initialization will occur.

Priorities in the range 0-20 are reserved for the C++ compiler. This is to ensure that proper initialization of the C++ run-time system takes place before the execution of your program. The "library" keyword represents a priority of 32 and can be used for class libraries that require initialization before the program is initialized. The "program" keyword represents a priority of 64 and is the default priority for any compiled code. Specifying "before" adjusts the priority by subtracting one. Specifying "after" adjusts the priority by adding one.

A source file containing the following "initialize" pragma specifies that the initialization of static data in the file will take place before initialization of all other static data in the program since a priority of 63 will be assigned.

Example:

```
#pragma initialize before program
```

If we specify "after" instead of "before", the initialization of the static data in the file will occur after initialization of all other static data in the program since a priority of 65 will be assigned.

Note that the following is equivalent to the "before" example

Example:

```
#pragma initialize 63
```

and the following is equivalent to the "after" example.

Example:

```
#pragma initialize 65
```

The use of the "before", "after", and "program" keywords are more descriptive in the intent of the pragmas.

It is recommended that a priority of 32 (the priority used when the "library" keyword is specified) be used when developing class libraries. This will ensure that initialization of static data defined by the class library will take place before initialization of static data defined by the program. The following "initialize" pragma can be used to achieve this.

Example:

```
#pragma initialize library
```

8.16 The INLINE_DEPTH Pragma (C++ Only)

When an in-line function is called, the function call may be replaced by the in-line expansion for that function. This in-line expansion may include calls to other in-line functions which can also be expanded. The "inline_depth" pragma can be used to set the number of times this expansion of in-line functions will occur for a call.

The form of the "inline_depth" pragma is as follows.

```
#pragma inline_depth [( n )] [;]
```

where ***description:***

n is the depth of expansion. If n is 0, no expansion will occur. If n is 1, only the original call is expanded. If n is 2, the original call and the in-line functions invoked by the original function will be expanded. The default value for n is 3. The maximum value for n is 255. Note that no expansion of recursive in-line functions occur unless enabled using the "inline_recursion" pragma.

8.17 The INLINE_RECURSION Pragma (C++ Only)

The "inline_recursion" pragma controls the recursive expansion of inline functions. The form of the "inline_recursion" pragma is as follows.

```
#pragma inline_recursion [(] on | off [)] [;]
```

Specifying "on" will enable expansion of recursive inline functions. The depth of expansion is specified by the "inline_depth" pragma. The default depth is 3. Specifying "off" suppresses expansion of recursive inline functions. This is the default.

8.18 The INTRINSIC Pragma

Certain functions, those listed in the description of the "oi" option, have intrinsic forms. These functions are special functions that are recognized by the compiler and processed in a special way. For example, the compiler may choose to generate in-line code for the function. The intrinsic attribute for these special functions is set by specifying the "oi" option or using an "intrinsic" pragma.

The following describes the form of the "intrinsic" pragma.

```
#pragma intrinsic ( fn {, fn} ) [;]
```

where ***description:***

fn is the name of a function.

Suppose the following source code was compiled without using the "oi" option so that no function had the intrinsic attribute. If we wanted the intrinsic form of the `sin` function to be used, we could specify the function in an "intrinsic" pragma.

```
#include <math.h>
#pragma intrinsic( sin );

double test( double x )
{
    return( sin( x ) );
}
```

8.19 The MESSAGE Pragma

The "message" pragma can be used to issue a message with the specified text to the standard output without terminating compilation. The following describes the form of the "message" pragma.

```
#pragma message ( "message text" ) [;]
```

where description:

"message text" is the text of the message that you wish to display.

The following is an example.

```
#if defined(__386__)
    ...
#else
#pragma message ( "assuming 16-bit compile" );
#endif
```

8.20 The ONCE Pragma

The "once" pragma can be used to indicate that the file which contains this pragma should only be opened and processed "once". The following describes the form of the "once" pragma.

```
#pragma once [;]
```

Assume that the file "foo.h" contains the following text.

Example:

```
#ifndef _FOO_H_INCLUDED
#define _FOO_H_INCLUDED
#pragma once
    .
    .
    .
#endif
```

The first time that the compiler processes "foo.h" and encounters the "once" pragma, it records the file's name. Subsequently, whenever the compiler encounters a `#include` statement that refers to "foo.h", it will not open the include file again. This can help speed up processing of `#include` files and reduce the time required to compile an application.

8.21 The PACK Pragma

The "pack" pragma can be used to control the way in which structures are stored in memory. By default, Watcom C/C++ aligns all structures and its fields on a byte boundary. There are 4 forms of the "pack" pragma.

The following form of the "pack" pragma can be used to change the alignment of structures and their fields in memory.

```
#pragma pack ( n ) [;]
```

where ***description:***

n is 1, 2, 4, 8 or 16 and specifies the method of alignment.

The alignment of structure members is described in the following table. If the size of the member is 1, 2, 4, 8 or 16, the alignment is given for each of the "zp" options. If the member of the structure is an array or structure, the alignment is described by the row "x".

sizeof(member)	zp1	zp2	zp4	zp8	zp16
1	0	0	0	0	0
2	0	2	2	2	2
4	0	2	4	4	4
8	0	2	4	8	8
16	0	2	4	8	16
x	aligned to largest member				

An alignment of 0 means no alignment, 2 means word boundary, 4 means doubleword boundary, etc. If the largest member of structure "x" is 1 byte then "x" is not aligned. If the largest member of structure "x" is 2 bytes then "x" is aligned according to row 2. If the largest member of structure "x" is 4 bytes then "x" is aligned according to row 4. If the largest member of structure "x" is 8 bytes then "x" is aligned according to row 8. If the largest member of structure "x" is 16 bytes then "x" is aligned according to row 16.

If no value is specified in the "pack" pragma, a default value of 2 is used. Note that the default value can be changed with the "zp" Watcom C/C++ compiler command line option.

The following form of the "pack" pragma can be used to save the current alignment amount on an internal stack.

```
#pragma pack ( push ) [;]
```

The following form of the "pack" pragma can be used to save the current alignment amount on an internal stack and set the current alignment.

```
#pragma pack ( push, number ) [;]
```

The following form of the "pack" pragma can be used to restore the previous alignment amount from an internal stack.

```
#pragma pack ( pop ) [;]
```

8.22 The READ_ONLY_FILE Pragma

Explicit listing of dependencies in a makefile can often be tedious in the development and maintenance phases of a project. The Watcom C/C++ compiler will insert dependency information into the object file as it processes source files so that a complete snapshot of the files necessary to build the object file are recorded. The "read_only_file" pragma can be used to prevent the name of the source file that includes it from being included in the dependency information that is written to the object file.

This pragma is commonly used in system header files since they change infrequently (and, when they do, there should be no impact on source files that have included them).

The form of the "read_only_file" pragma follows.

```
#pragma read_only_file [;]
```

For more information on make dependencies, see the section entitled "Automatic Dependency Detection (.AUTODEPEND)" in the ***Watcom C/C++ Tools User's Guide.***

8.23 The TEMPLATE_DEPTH Pragma (C++ Only)

The "template_depth" pragma provides a hard limit for the amount of nested template expansions allowed so that infinite expansion can be detected.

The form of the "template_depth" pragma is as follows.

```
#pragma template_depth [(] n [)] [;]
```

where ***description:***

n is the depth of expansion. If the value of n is less than 2, if will default to 2. If n is not specified, a warning message will be issued and the default value for n will be 100.

The following example of recursive template expansion illustrates why this pragma can be useful.

Example:

```
#pragma template_depth(10);

template <class T>
struct S {
    S<T*> x;
};

S<char> v;
```

8.24 The WARNING Pragma (C++ Only)

The "warning" pragma sets the level of warning messages. The form of the "warning" pragma is as follows.

```
#pragma warning msg_num level [;]
```

where ***description:***

msg_num is the number of the warning message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Watcom C++ Diagnostic Messages" on page 397. If `msg_num` is "*", the level of all warning messages is changed to the specified level. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

level is a number from 0 to 9 and represents the level of the warning message. When a value of zero is specified, the warning becomes an error.

8.25 Auxiliary Pragmas

The following sections describe the capabilities provided by auxiliary pragmas.

8.25.1 Specifying Symbol Attributes

Auxiliary pragmas are used to describe attributes that affect code generation. Initially, the compiler defines a default set of attributes. Each auxiliary pragma refers to one of the following.

1. a symbol (such as a variable or function)
2. a type definition that resolves to a function type
3. the default set of attributes defined by the compiler

When an auxiliary pragma refers to a particular symbol, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to the specified symbol and can only be changed by another auxiliary pragma that refers to the same symbol.

An example of a type definition that resolves to a function type is the following.

```
typedef void (*func_type)();
```

When an auxiliary pragma refers to a such a type definition, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to each function whose type matches the specified type definition.

When "default" is specified instead of a symbol name, the attributes specified by the auxiliary pragma change the default set of attributes. The resulting attributes are used by all symbols that have not been specifically referenced by a previous auxiliary pragma.

Note that all auxiliary pragmas are processed before code generation begins. Consider the following example.

```
code in which symbol x is referenced
#pragma aux y <attrs_1>;
code in which symbol y is referenced
code in which symbol z is referenced
#pragma aux default <attrs_2>;
#pragma aux x <attrs_3>;
```

Auxiliary attributes are assigned to `x, y` and `z` in the following way.

1. Symbol `x` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>` and `<attrs_3>.`

2. Symbol `y` is assigned the initial default attributes merged with the attributes specified by `<attrs_1>.`

3. Symbol `z` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>.`

8.25.2 Alias Names

When a symbol referred to by an auxiliary pragma includes an alias name, the attributes of the alias name are also assumed by the specified symbol.

There are two methods of specifying alias information. In the first method, the symbol assumes only the attributes of the alias name; no additional attributes can be specified. The second method is more general since it is possible to specify an alias name as well as additional auxiliary information. In this case, the symbol assumes the attributes of the alias name as well as the attributes specified by the additional auxiliary information.

The simple form of the auxiliary pragma used to specify an alias is as follows.

```
#pragma aux ( sym, alias ) [;]
```

where ***description:***

sym is any valid C/C++ identifier.

alias is the alias name and is any valid C/C++ identifier.

Consider the following example.

```
#pragma aux push_args parm [] ;
#pragma aux ( rtn, push_args ) ;
```

The routine `rtn` assumes the attributes of the alias name `push_args` which specifies that the arguments to `rtn` are passed on the stack.

Let us look at an example in which the symbol is a type definition.

```
typedef void (func_type)(int);

#pragma aux push_args parm [];
#pragma aux ( func_type, push_args );

extern func_type rtn1;
extern func_type rtn2;
```

The first auxiliary pragma defines an alias name called `push_args` that specifies the mechanism to be used to pass arguments. The mechanism is to pass all arguments on the stack. The second auxiliary pragma associates the attributes specified in the first pragma with the type definition `func_type.` Since `rtn1` and `rtn2` are of type `func_type,` arguments to either of those functions will be passed on the stack.

The general form of an auxiliary pragma that can be used to specify an alias is as follows.

```
#pragma aux ( alias ) sym aux_attrs [;]
```

where ***description:***

alias is the alias name and is any valid C/C++ identifier.

sym is any valid C/C++ identifier.

aux_attrs are attributes that can be specified with the auxiliary pragma.

Consider the following example.

```
#pragma aux MS_C "_*"                                   \
                 parm caller []                         \
                 value struct float struct routine [ax]\
                 modify [ax bx cx dx es];
#pragma aux (MS_C) rtn1;
#pragma aux (MS_C) rtn2;
#pragma aux (MS_C) rtn3;
```

The routines `rtn1, rtn2` and `rtn3` assume the same attributes as the alias name `MS_C` which defines the calling convention used by the Microsoft C compiler. Whenever calls are made to `rtn1, rtn2` and `rtn3,` the Microsoft C calling convention will be used.

Note that if the attributes of `MS_C` change, only one pragma needs to be changed. If we had not used an alias name and specified the attributes in each of the three pragmas for `rtn1, rtn2` and `rtn3,` we would have to change all three pragmas. This approach also reduces the amount of memory required by the compiler to process the source file.

> ***WARNING!*** The alias name `MS_C` is just another symbol. If `MS_C` appeared in your source code, it would assume the attributes specified in the pragma for `MS_C.`

8.25.3 Predefined Aliases

A number of symbols are predefined by the compiler with a set of attributes that describe a particular calling convention. These symbols can be used as aliases. The following is a list of these symbols.

__cdecl `__cdecl` or `cdecl` defines the calling convention used by Microsoft compilers.

__pascal `__pascal` or `pascal` defines the calling convention used by OS/2 1.x and Windows 3.x API functions.

The following describes the attributes of the above alias names.

8.25.3.1 Predefined "__cdecl" Alias

```
#pragma aux __cdecl "_*" \
            parm caller [] \
            value struct float struct routine [ax] \
            modify [ax bx cx dx es]
```

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine allocates space for the return value and returns a pointer to the return value in register AX.

4. Registers AX, BX, CX and DX, and segment register ES are not saved and restored when a call is made.

8.25.3.2 Predefined "__pascal" Alias

```
#pragma aux __pascal "^" \
            parm reverse routine [] \
            value struct float struct caller [] \
            modify [ax bx cx dx es]
```

Notes:

1. All symbols are mapped to upper case.

2. Arguments are pushed on the stack in reverse order. That is, the first argument is pushed first, the second argument is pushed next, and so on. The routine being called will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register AX will contain address of the space allocated for the return value.

4. Registers AX, BX, CX and DX, and segment register ES are not saved and restored when a call is made.

8.25.4 Alternate Names for Symbols

The following form of the auxiliary pragma can be used to describe the mapping of a symbol from its source form to its object form.

```
#pragma aux sym obj_name [;]
```

where ***description:***

sym is any valid C/C++ identifier.

obj_name is any character string enclosed in double quotes.

When specifying `obj_name,` the asterisk character (’*’) has a special meaning; it is a placeholder for `sym.`

In the following example, the name "myrtn" will be replaced by "myrtn_" in the object file.

```
#pragma aux myrtn "*_";
```

This is the default for all function names.

In the following example, the name "myvar" will be replaced by "_myvar" in the object file.

```
#pragma aux myvar "_*";
```

This is the default for all variable names.

The default mapping for all symbols can also be changed as illustrated by the following example.

```
#pragma aux default "_*_";
```

The above auxiliary pragma specifies that all names will be prefixed and suffixed by an underscore character ('_').

The '^' character also has a special meaning. Whenever it is encountered in `obj_name,` it is replaced by the upper case version of `sym.`

In the following example, the name "myrtn" will be replaced by "MYRTN" in the object file.

```
#pragma aux myrtn "^";
```

8.25.5 Describing Calling Information

The following form of the auxiliary pragma can be used to describe the way a function is to be called.

```
#pragma aux sym far [;]
    or
#pragma aux sym near [;]
    or
#pragma aux sym = in_line [;]

in_line ::= { const | (seg id) | (offset id) | (reloff id)
          | (float fpinst) | "asm" }
```

where	*description:*
sym	is a function name.
const	is a valid C/C++ integer constant.
id	is any valid C/C++ identifier.
fpinst	is a sequence of bytes that forms a valid 80x87 instruction. The keyword ***float*** must precede `fpinst` so that special fixups are applied to the 80x87 instruction.
seg	specifies the segment of the symbol `id.`
offset	specifies the offset of the symbol `id.`

reloff specifies the relative offset of the symbol `id` for near control transfers.

asm is an assembly language instruction or directive.

In the following example, Watcom C/C++ will generate a far call to the function `myrtn.`

```
#pragma aux myrtn far;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a far call will be generated even if you are compiling for a memory model with a small code model.

In the following example, Watcom C/C++ will generate a near call to the function `myrtn.`

```
#pragma aux myrtn near;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a near call will be generated even if you are compiling for a memory model with a big code model.

In the following DOS example, Watcom C/C++ will generate the sequence of bytes following the "=" character in the auxiliary pragma whenever a call to `mode4` is encountered. `mode4` is called an in-line function.

```
void mode4(void);
#pragma aux mode4 =                 \
    0xb4 0x00       /* mov AH,0 */ \
    0xb0 0x04       /* mov AL,4 */ \
    0xcd 0x10       /* int 10H  */ \
    modify [ AH AL ];
```

The sequence in the above DOS example represents the following lines of assembly language instructions.

```
mov   AH,0       ; select function "set mode"
mov   AL,4       ; specify mode (mode 4)
int   10H        ; BIOS video call
```

The above example demonstrates how to generate BIOS function calls in-line without writing an assembly language function and calling it from your C/C++ program. The C prototype for the function `mode4` is not necessary but is included so that we can take advantage of the argument type checking provided by Watcom C/C++.

The following DOS example is equivalent to the above example but mnemonics for the assembly language instructions are used instead of the binary encoding of the assembly language instructions.

```
void mode4(void);
#pragma aux mode4 =     \
    "mov AH,0",         \
    "mov AL,4",         \
    "int 10H"           \
    modify [ AH AL ];
```

If a sequence of in-line assembly language instructions contains 80x87 floating-point instructions, each floating-point instruction must be preceded by "float". Note that this is only required if you have specified the "fpi" compiler option; otherwise it will be ignored.

The following example generates the 80x87 "square root" instruction.

```
double mysqrt(double);
#pragma aux mysqrt parm [8087] = \
    float 0xd9 0xfa /* fsqrt */;
```

A sequence of in-line assembly language instructions may contain symbolic references. In the following example, a near call to the function `myalias` is made whenever `myrtn` is called.

```
extern void myalias(void);
void myrtn(void);
#pragma aux myrtn =                     \
    0xe8 offset myalias /* near call */;
```

In the following example, a far call to the function `myalias` is made whenever `myrtn` is called.

```
extern void myalias(void);
void myrtn(void);
#pragma aux myrtn =                                 \
    0x9a offset myalias seg myalias /* far call */;
```

8.25.5.1 Loading Data Segment Register

An application may have been compiled so that the segment register DS does not contain the segment address of the default data segment (group "DGROUP"). This is usually the case if you are using a large data memory model. Suppose you wish to call a function that assumes that the segment register DS contains the segment address of the default data segment. It would be very cumbersome if you were forced to compile your application so that the segment register DS contained the default data segment (a small data memory model).

The following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment before calling the specified function.

```
#pragma aux sym parm loadds [;]
```

where ***description:***

sym is a function name.

Alternatively, the following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment as part of the prologue sequence for the specified function.

```
#pragma aux sym loadds [;]
```

where ***description:***

sym is a function name.

8.25.5.2 Defining Exported Symbols in Dynamic Link Libraries

An exported symbol in a dynamic link library is a symbol that can be referenced by an application that is linked with that dynamic link library. Normally, symbols in dynamic link libraries are exported using the Watcom Linker "EXPORT" directive. An alternative method is to use the following form of the auxiliary pragma.

```
#pragma aux sym export [;]
```

where ***description:***

sym is a function name.

8.25.5.3 Defining Windows Callback Functions

When compiling a Microsoft Windows application, you must use the "zW" option so that special prologue/epilogue sequences are generated. Furthermore, callback functions require

larger prologue/epilogue sequences than those generated when the "zW" compiler option is specified. The following form of the auxiliary pragma will cause a callback prologue/epilogue sequence to be generated for a callback function when compiled using the "zW" option.

```
#pragma aux sym export [;]
```

where ***description:***

sym is a callback function name.

Alternatively, the "zw" compiler option can be used to generate callback prologue/epilogue sequences. However, all functions contained in a module compiled using the "zw" option will have a callback prologue/epilogue sequence even if the functions are not callback functions.

8.25.5.4 Forcing a Stack Frame

Normally, a function contains a stack frame if arguments are passed on the stack or an automatic variable is allocated on the stack. No stack frame will be generated if the above conditions are not satisfied. The following form of the auxiliary pragma will force a stack frame to be generated under any circumstance.

```
#pragma aux sym frame [;]
```

where ***description:***

sym is a function name.

8.25.6 Describing Argument Information

Using auxiliary pragmas, you can describe the calling convention that Watcom C/C++ is to use for calling functions. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes argument passing is the following.

```
#pragma aux sym parm { pop_info | reverse | {reg_set} } [;]

pop_info ::= caller | routine
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

8.25.6.1 Passing Arguments in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to pass arguments to a particular function.

```
#pragma aux sym parm {reg_set} [;]
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

Register sets establish a priority for register allocation during argument list processing. Register sets are processed from left to right. However, within a register set, registers are chosen in any order. Once all register sets have been processed, any remaining arguments are pushed on the stack.

Note that regardless of the register sets specified, only certain combinations of registers will be selected for arguments of a particular type.

Note that arguments of type **float** and **double** are always pushed on the stack when the "fpi" or "fpi87" option is used.

double Arguments of type **double** can only be passed in the following register combination: AX:BX:CX:DX. For example, if the following register set was specified for a routine having an argument of type **double,**

```
[AX BX SI DI]
```

the argument would be pushed on the stack since a valid register combination for 8-byte arguments is not contained in the register set. Note that this method for passing arguments of type **double** is supported only when the "fpc" option is used. Note that this argument passing method does not include the passing of 8-byte structures.

far pointer A far pointer can only be passed in one of the following register pairs: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX, BX:AX, DS:CX, DS:DX, DS:DI, DS:SI, DS:BX, DS:AX, ES:CX, ES:DX, ES:DI, ES:SI, ES:BX or ES:AX. For example, if a far pointer is passed to a function with the following register set,

```
[ES BP]
```

the argument would be pushed on the stack since a valid register combination for a far pointer is not contained in the register set.

long int, float

The only registers that will be assigned to 4-byte arguments (e.g., arguments of type **long int,**) are: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX and BX:AX. For example, if the following register set was specified for a routine with one argument of type **long int,**

```
[ES DI]
```

the argument would be pushed on the stack since a valid register combination for 4-byte arguments is not contained in the register set. Note that this argument passing method includes 4-byte structures. Note that this argument passing method includes arguments of type **float** but only when the "fpc" option is used.

int The only registers that will be assigned to 2-byte arguments (e.g., arguments of type **int**) are: AX, BX, CX, DX, SI and DI. For example, if the following register set was specified for a routine with one argument of type **int,**

```
[BP]
```

the argument would be pushed on the stack since a valid register combination for 2-byte arguments is not contained in the register set.

char Arguments whose size is 1 byte (e.g., arguments of type **char**) are promoted to 2 bytes and are then assigned registers as if they were 2-byte arguments.

others Arguments that do not fall into one of the above categories cannot be passed in registers and are pushed on the stack. Once an argument has been assigned a position on the stack, all remaining arguments will be assigned a position on the stack even if all register sets have not yet been exhausted.

Notes:

1. The default register set is [AX BX CX DX].

2. Specifying registers AH and AL is equivalent to specifying register AX. Specifying registers DH and DL is equivalent to specifying register DX. Specifying registers CH and CL is equivalent to specifying register CX. Specifying registers BH and BL is equivalent to specifying register BX.

3. If you are compiling for a memory model with a small data model, or the "zdp" compiler option is specified, any register combination containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment. Note that the "zdf" compiler option can be used to specify that register DS does not contain that segment address of the program's data segment. In this case, register combinations containing register DS are legal.

Consider the following example.

```
#pragma aux myrtn parm [ax bx cx dx] [bp si];
```

Suppose `myrtn` is a routine with 3 arguments each of type **long int.**

1. The first argument will be passed in the register pair DX:AX.
2. The second argument will be passed in the register pair CX:BX.
3. The third argument will be pushed on the stack since BP:SI is not a valid register pair for arguments of type **long int.**

It is possible for registers from the second register set to be used before registers from the first register set are used. Consider the following example.

```
#pragma aux myrtn parm [ax bx cx dx] [si di];
```

Suppose `myrtn` is a routine with 3 arguments, the first of type **int** and the second and third of type **long int.**

1. The first argument will be passed in the register AX.
2. The second argument will be passed in the register pair CX:BX.
3. The third argument will be passed in the register set DI:SI.

Note that registers are no longer selected from a register set after registers are selected from subsequent register sets, even if all registers from the original register set have not been exhausted.

An empty register set is permitted. All subsequent register sets appearing after an empty register set are ignored; all remaining arguments are pushed on the stack.

Notes:

1. If a single empty register set is specified, all arguments are passed on the stack.

2. If no register set is specified, the default register set [AX BX CX DX] is used.

8.25.6.2 Forcing Arguments into Specific Registers

It is possible to force arguments into specific registers. Suppose you have a function, say "mycopy", that copies data. The first argument is the source, the second argument is the destination, and the third argument is the length to copy. If we want the first argument to be passed in the register SI, the second argument to be passed in register DI and the third argument to be passed in register CX, the following auxiliary pragma can be used.

```
void mycopy( char near *, char *, int );
#pragma aux mycopy parm [DI] [SI] [CX];
```

Note that you must be aware of the size of the arguments to ensure that the arguments get passed in the appropriate registers.

8.25.6.3 Passing Arguments to In-Line Functions

For functions whose code is generated by Watcom C/C++ and whose argument list is described by an auxiliary pragma, Watcom C/C++ has some freedom in choosing how arguments are assigned to registers. Since the code for in-line functions is specified by the programmer, the description of the argument list must be very explicit. To achieve this, Watcom C/C++ assumes that each register set corresponds to an argument. Consider the following DOS example of an in-line function called `scrollactivepgup.`

```
void scrollactivepgup(char,char,char,char,char,char);
#pragma aux scrollactivepgup = \
    "mov AH,6"   \
    "int 10h"    \
    parm [ch] [cl] [dh] [dl] [al] [bh] \
    modify [ah];
```

The BIOS video call to scroll the active page up requires the following arguments.

1. The row and column of the upper left corner of the scroll window is passed in registers CH and CL respectively.

2. The row and column of the lower right corner of the scroll window is passed in registers DH and DL respectively.

3. The number of lines blanked at the bottom of the window is passed in register AL.

4. The attribute to be used on the blank lines is passed in register BH.

When passing arguments, Watcom C/C++ will convert the argument so that it fits in the register(s) specified in the register set for that argument. For example, in the above example, if the first argument to `scrollactivepgup` was called with an argument whose type was **int**, it would first be converted to **char** before assigning it to register CH. Similarly, if an in-line function required its argument in register pair DX:AX and the argument was of type **short int**, the argument would be converted to **long int** before assigning it to register pair DX:AX.

In general, Watcom C/C++ assigns the following types to register sets.

1. A register set consisting of a single 8-bit register (1 byte) is assigned a type of **unsigned char.**

2. A register set consisting of a single 16-bit register (2 bytes) is assigned a type of **unsigned short int.**

3. A register set consisting of two 16-bit registers (4 bytes) is assigned a type of **unsigned long int.**

4. A register set consisting of four 16-bit registers (8 bytes) is assigned a type of **double.**

8.25.6.4 Removing Arguments from the Stack

The following form of the auxiliary pragma specifies who removes from the stack arguments that were pushed on the stack.

```
#pragma aux sym parm (caller | routine) [;]
```

where ***description:***

sym is a function name.

"caller" specifies that the caller will pop the arguments from the stack; "routine" specifies that the called routine will pop the arguments from the stack. If "caller" or "routine" is omitted, "routine" is assumed unless the default has been changed in a previous auxiliary pragma, in which case the new default is assumed.

8.25.6.5 Passing Arguments in Reverse Order

The following form of the auxiliary pragma specifies that arguments are passed in the reverse order.

```
#pragma aux sym parm reverse [;]
```

where ***description:***

sym is a function name.

Normally, arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left.

When arguments are reversed, the rightmost arguments are passed in registers and the leftmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from left to right.

Reversing arguments is most useful for functions that require arguments to be passed on the stack in an order opposite from the default. The following auxiliary pragma demonstrates such a function.

```
#pragma aux rtn parm reverse [];
```

8.25.7 Describing Function Return Information

Using auxiliary pragmas, you can describe the way functions are to return values. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes the way a function returns its value is the following.

```
#pragma aux sym value {no8087 | reg_set | struct_info} [;]
struct_info ::= struct {float | struct | (routine | caller) | reg_set}
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

8.25.7.1 Returning Function Values in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to return a function's value.

```
#pragma aux sym value reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

Note that the method described below for returning values of type **float** or **double** is supported only when the "fpc" option is used.

Depending on the type of the return value, only certain registers are allowed in *reg_set*.

1-byte For 1-byte return values, only the following registers are allowed: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2-byte For 2-byte return values, only the following registers are allowed: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

4-byte For 4-byte return values (except far pointers), only the following register pairs are allowed: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX or BX:AX. If no register set is specified, registers DX:AX will be used. This form of the auxiliary pragma is legal for functions of type **float** when using the "fpc" option only.

far pointer For functions that return far pointers, the following register pairs are allowed: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX, BX:AX, DS:CX, DS:DX, DS:DI, DS:SI, DS:BX, DS:AX, ES:CX, ES:DX, ES:DI, ES:SI, ES:BX or ES:AX. If no register set is specified, the registers DX:AX will be used.

8-byte For 8-byte return values (including functions of type **double**), only the following register combination is allowed: AX:BX:CX:DX. If no register set is specified, the registers AX:BX:CX:DX will be used. This form of the auxiliary pragma is legal for functions of type **double** when using the "fpc" option only.

Notes:

1. An empty register set is not allowed.

2. If you are compiling for a memory model which has a small data model, any of the above register combinations containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program’s data segment.

8.25.7.2 Returning Structures

Typically, structures are not returned in registers. Instead, the caller allocates space on the stack for the return value and sets register SI to point to it. The called routine then places the return value at the location pointed to by register SI.

The following form of the auxiliary pragma can be used to specify the register that is to be used to point to the return value.

```
#pragma aux sym value struct (caller|routine) reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

"caller" specifies that the caller will allocate memory for the return value. The address of the memory allocated for the return value is placed in the register specified in the register set by the caller before the function is called. If an empty register set is specified, the address of the memory allocated for the return value will be pushed on the stack immediately before the call and will be returned in register AX by the called routine. It is assumed that the memory for the return value is allocated from the stack segment (the stack segment is contained in segment register SS).

"routine" specifies that the called routine will allocate memory for the return value. Upon returning to the caller, the register specified in the register set will contain the address of the return value. An empty register set is not allowed.

Only the following registers are allowed in the register set: AX, DX, BX, CX, SI or DI. Note that in a big data model, the address in the return register is assumed to be in the segment specified by the value in the SS segment register.

If the size of the structure being returned is 1, 2 or 4 bytes, it will be returned in registers. The return register will be selected from the register set in the following way.

1. A 1-byte structure will be returned in one of the following registers: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2. A 2-byte structure will be returned in one of the following registers: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

3. A 4-byte structure will be returned in one of the following register pairs: DX:AX, CX:BX, CX:AX, CX:SI, DX:BX, DI:AX, CX:DI, DX:SI, DI:BX, SI:AX, CX:DX, DX:DI, DI:SI, SI:BX or BX:AX. If no register set is specified, register pair DX:AX will be used.

The following form of the auxiliary pragma can be used to specify that structures whose size is 1, 2 or 4 bytes are not to be returned in registers. Instead, the caller will allocate space on the stack for the structure return value and point register SI to it.

```
#pragma aux sym value struct struct [;]
```

where ***description:***

sym is a function name.

8.25.7.3 Returning Floating-Point Data

There are a few ways available for specifying how the value for a function whose type is **float** or **double** is to be returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in registers. Instead, the caller will allocate space on the stack for the return value and point register SI to it.

```
#pragma aux sym value struct float [;]
```

where ***description:***

sym is a function name.

In other words, floating-point values are to be returned in the same way structures are returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in 80x87 registers when compiling with the "fpi" or "fpi87" option. Instead, the value will be returned in 80x86 registers. This is the default behaviour for the "fpc" option. Function return values whose type is **float** will be returned in registers DX:AX. Function return values whose type is **double** will be returned in registers AX:BX:CX:DX. This is the default method for the "fpc" option.

```
#pragma aux sym value no8087 [;]
```

where ***description:***

sym is a function name.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are to be returned in ST(0) when compiling with the "fpi" or "fpi87" option. This form of the auxiliary pragma is not legal for the "fpc" option.

```
#pragma aux sym value [8087] [;]
```

where ***description:***

sym is a function name.

8.25.8 A Function that Never Returns

The following form of the auxiliary pragma can be used to describe a function that does not return to the caller.

```
#pragma aux sym aborts [;]
```

where ***description:***

sym is a function name.

Consider the following example.

```
#pragma aux exitrtn aborts;
extern void exitrtn(void);

void rtn()
  {
    exitrtn();
  }
```

`exitrtn` is defined to be a function that does not return. For example, it may call `exit` to return to the system. In this case, Watcom C/C++ generates a "jmp" instruction instead of a "call" instruction to invoke `exitrtn.`

8.25.9 Describing How Functions Use Memory

The following form of the auxiliary pragma can be used to describe a function that does not modify any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux sym modify nomemory [;]
```

where ***description:***

sym is a function name.

Consider the following example.

```
#pragma off (check_stack);

extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
    while( i < 10000 ) {
        i += 383;
    }
    myrtn();
    i += 13143;
};
```

To compile the above program, "rtn.c", we issue the following command.

```
C>wcc rtn /oai /d1
C>wpp rtn /oai /d1
C>wcc386 rtn /oai /d1
C>wpp386 rtn /oai /d1
```

For illustrative purposes, we omit loop optimizations from the list of code optimizations that we want the compiler to perform. The "d1" compiler option is specified so that the object file produced by Watcom C/C++ contains source line information.

We can generate a file containing a disassembly of `RTN.OBJ` by issuing the following command.

```
C>wdis rtn /l /s /r
```

The "s" option is specified so that the listing file produced by the Watcom Disassembler contains source lines taken from RTN.C. The listing file RTN.LST appears as follows.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE  0026 bytes

#pragma off (check_stack);

extern void MyRtn( void );

int i = { 1033 };

extern Rtn()
  {
 0000  52                Rtn_            push    DX
 0001  8b 16 00 00                       mov     DX,_i

    while( i < 10000 ) {
 0005  81 fa 10 27       L1              cmp     DX,2710H
 0009  7d 06                             jge     L2

        i += 383;
    }
 000b  81 c2 7f 01                       add     DX,017fH
 000f  eb f4                             jmp     L1

    MyRtn();
 0011  89 16 00 00       L2              mov     _i,DX
 0015  e8 00 00                          call    MyRtn_
 0018  8b 16 00 00                       mov     DX,_i

    i += 13143;
 001c  81 c2 57 33                       add     DX,3357H
 0020  89 16 00 00                       mov     _i,DX

  };
 0024  5a                                pop     DX
 0025  c3                                ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD  0002 bytes
 0000  09 04                    _i              - ..

No disassembly errors

------------------------------------------------------------
```

Let us add the following auxiliary pragma to the source file.

```
#pragma aux myrtn modify nomemory;
```

If we compile the source file with the above pragma and disassemble the object file using the Watcom Disassembler, we get the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE  0022 bytes

#pragma off (check_stack);

extern void MyRtn( void );
#pragma aux MyRtn modify nomemory;

int i = { 1033 };

extern Rtn()
  {
 0000  52                Rtn_            push    DX
 0001  8b 16 00 00                       mov     DX,_i

    while( i < 10000 ) {
 0005  81 fa 10 27       L1              cmp     DX,2710H
 0009  7d 06                             jge     L2

        i += 383;
    }
 000b  81 c2 7f 01                       add     DX,017fH
 000f  eb f4                             jmp     L1

    MyRtn();
 0011  89 16 00 00       L2              mov     _i,DX
 0015  e8 00 00                          call    MyRtn_

    i += 13143;
 0018  81 c2 57 33                       add     DX,3357H
 001c  89 16 00 00                       mov     _i,DX

 };
 0020  5a                                pop     DX
 0021  c3                                ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD  0002 bytes
 0000  09 04                    _i              - ..

No disassembly errors

------------------------------------------------------------
```

Notice that the value of `i` is in register DX after completion of the "while" loop. After the call to `myrtn,` the value of `i` is not loaded from memory into a register to perform the final addition. The auxiliary pragma informs the compiler that `myrtn` does not modify any

memory (i.e., global or static variables) that is used directly or indirectly by `Rtn` and hence register DX contains the correct value of `i.`

The preceding auxiliary pragma deals with routines that modify memory. Let us consider the case where routines reference memory. The following form of the auxiliary pragma can be used to describe a function that does not reference any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux sym parm nomemory modify nomemory [;]
```

where ***description:***

sym is a function name.

Notes:

1. You must specify both "parm nomemory" and "modify nomemory".

Let us replace the auxiliary pragma in the above example with the following auxiliary pragma.

```
#pragma aux myrtn parm nomemory modify nomemory;
```

If you now compile our source file and disassemble the object file using WDIS, the result is the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE  001e bytes

#pragma off (check_stack);

extern void MyRtn( void );
#pragma aux MyRtn parm nomemory modify nomemory;

int i = { 1033 };
```

```
extern Rtn()
  {
 0000  52                Rtn_            push    DX
 0001  8b 16 00 00                        mov     DX,_i

    while( i < 10000 ) {
 0005  81 fa 10 27       L1              cmp     DX,2710H
 0009  7d 06                             jge     L2

        i += 383;
    }
 000b  81 c2 7f 01                       add     DX,017fH
 000f  eb f4                             jmp     L1

    MyRtn();
 0011  e8 00 00          L2              call    MyRtn_

    i += 13143;
 0014  81 c2 57 33                       add     DX,3357H
 0018  89 16 00 00                       mov     _i,DX

 };
 001c  5a                                pop     DX
 001d  c3                                ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD  0002 bytes
 0000  09 04                    _i              - ..

No disassembly errors

------------------------------------------------------------
```

Notice that after completion of the "while" loop we did not have to update `i` with the value in register DX before calling `myrtn.` The auxiliary pragma informs the compiler that `myrtn` does not reference any memory (i.e., global or static variables) that is used directly or indirectly by `myrtn` so updating `i` was not necessary before calling `myrtn.`

8.25.10 Describing the Registers Modified by a Function

The following form of the auxiliary pragma can be used to describe the registers that a function will use without saving.

```
#pragma aux sym modify [exact] reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

Specifying a register set informs Watcom C/C++ that the registers belonging to the register set are modified by the function. That is, the value in a register before calling the function is different from its value after execution of the function.

Registers that are used to pass arguments are assumed to be modified and hence do not have to be saved and restored by the called function. Also, since the AX register is frequently used to return a value, it is always assumed to be modified. If necessary, the caller will contain code to save and restore the contents of registers used to pass arguments. Note that saving and restoring the contents of these registers may not be necessary if the called function does not modify them. The following form of the auxiliary pragma can be used to describe exactly those registers that will be modified by the called function.

```
#pragma aux sym modify exact reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

The above form of the auxiliary pragma tells Watcom C/C++ not to assume that the registers used to pass arguments will be modified by the called function. Instead, only the registers specified in the register set will be modified. This will prevent generation of the code which unnecessarily saves and restores the contents of the registers used to pass arguments.

Also, any registers that are specified in the `value` register set are assumed to be unmodified unless explicitly listed in the `exact` register set. In the following example, the code generator will not generate code to save and restore the value of the stack pointer register since we have told it that "GetSP" does not modify any register whatsoever.

Example:

```
unsigned GetSP(void);
#if defined(__386__)
#pragma aux GetSP = value [esp] modify exact [];
#else
#pragma aux GetSP = value [sp] modify exact [];
#endif
```

8.25.11 An Example

As mentioned in an earlier section, the following pragma defines the calling convention for functions compiled by Microsoft C.

```
#pragma aux MS_C "_*"                                  \
                 parm caller []                        \
                 value struct float struct routine [ax]\
                 modify [ax bx cx dx es];
```

Let us discuss this pragma in detail.

"_"* specifies that all function and variable names are preceded by the underscore character (_) when translated from source form to object form.

parm caller [] specifies that all arguments are to be passed on the stack (an empty register set was specified) and the caller will remove the arguments from the stack.

value struct marks the section describing how the called routine returns structure information.

- ***float*** specifies that floating-point arguments are returned in the same way as structures are returned.
- ***struct*** specifies that 1, 2 and 4-byte structures are not to be returned in registers.
- ***routine*** specifies that the called routine allocates storage for the return structure and returns with a register pointing at it.
- ***[ax]*** specifies that register AX is used to point to the structure return value.

modify [ax bx cx dx es]

specifies that registers AX, BX, CX, DX and ES are not preserved by the called routine.

Note that the default method of returning integer values is used; 1-byte characters are returned in register AL, 2-byte integers are returned in register AX, and 4-byte integers are returned in the register pair DX:AX.

8.25.12 Auxiliary Pragmas and the 80x87

This section deals with those aspects of auxiliary pragmas that are specific to the 80x87. The discussion in this chapter assumes that one of the "fpi" or "fpi87" options is used to compile functions. The following areas are affected by the use of these options.

1. passing floating-point arguments to functions,
2. returning floating-point values from functions and
3. which 80x87 floating-point registers are allowed to be modified by the called routine.

8.25.12.1 Using the 80x87 to Pass Arguments

By default, floating-point arguments are passed on the 80x86 stack. The 80x86 registers are never used to pass floating-point arguments when a function is compiled with the "fpi" or "fpi87" option. However, they can be used to pass arguments whose type is not floating-point such as arguments of type "int".

The following form of the auxiliary pragma can be used to describe the registers that are to be used to pass arguments to functions.

```
#pragma aux sym parm {reg_set} [;]
```

where ***description:***

sym is a function name.

reg_set is a register set. The register set can contain 80x86 registers and/or the string "8087".

Notes:

1. If an empty register set is specified, all arguments, including floating-point arguments, will be passed on the 80x86 stack.

When the string "8087" appears in a register set, it simply means that floating-point arguments can be passed in 80x87 floating-point registers if the source file is compiled with the "fpi" or "fpi87" option. Before discussing argument passing in detail, some general notes on the use of the 80x87 floating-point registers are given.

The 80x87 contains 8 floating-point registers which essentially form a stack. The stack pointer is called ST and is a number between 0 and 7 identifying which 80x87 floating-point register is at the top of the stack. ST is initially 0. 80x87 instructions reference these registers by specifying a floating-point register number. This number is then added to the current value of ST. The sum (taken modulo 8) specifies the 80x87 floating-point register to be used. The notation ST(n), where "n" is between 0 and 7, is used to refer to the position of an 80x87 floating-point register relative to ST.

When a floating-point value is loaded onto the 80x87 floating-point register stack, ST is decremented (modulo 8), and the value is loaded into ST(0). When a floating-point value is stored and popped from the 80x87 floating-point register stack, ST is incremented (modulo 8) and ST(1) becomes ST(0). The following illustrates the use of the 80x87 floating-point registers as a stack, assuming that the value of ST is 4 (4 values have been loaded onto the 80x87 floating-point register stack).

```
              +-----------------+
        0     | 4th from top    |  ST(4)
              +-----------------+
        1     | 5th from top    |  ST(5)
              +-----------------+
        2     | 6th from top    |  ST(6)
              +-----------------+
        3     | 7th from top    |  ST(7)
              +-----------------+
ST -> 4     | top of stack    |  ST(0)
              +-----------------+
        5     | 1st from top    |  ST(1)
              +-----------------+
        6     | 2nd from top    |  ST(2)
              +-----------------+
        7     | 3rd from top    |  ST(3)
              +-----------------+
```

Starting with version 9.5, the Watcom compilers use all eight of the 80x87 registers as a stack. The initial state of the 80x87 register stack is empty before a program begins execution.

Note: For compatibility with code compiled with version 9.0 and earlier, you can compile with the "fpr" option. In this case only four of the eight 80x87 registers are used as a stack. These four registers were used to pass arguments. The other four registers form what was called the 80x87 cache. The cache was used for local floating-point variables. The state of the 80x87 registers before a program began execution was as follows.

1. The four 80x87 floating-point registers that form the stack are uninitialized.
2. The four 80x87 floating-point registers that form the 80x87 cache are initialized with zero.

Hence, initially the 80x87 cache was comprised of ST(0), ST(1), ST(2) and ST(3). ST had the value 4 as in the above diagram. When a floating-point value was pushed on the stack (as is the case when passing floating-point arguments), it became ST(0) and the 80x87 cache was comprised of ST(1), ST(2), ST(3) and ST(4). When the 80x87 stack was full, ST(0), ST(1), ST(2) and ST(3) formed the stack and ST(4), ST(5), ST(6) and ST(7) formed the 80x87 cache. Version 9.5 and later no longer use this strategy.

The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, and a previous argument has been assigned a position on the 80x86 stack (instead of the 80x87 stack), the floating-point argument is also assigned a position on the 80x86 stack. Otherwise proceed to the next step.

3. If the string "8087" appears in a register set in the pragma, and if the 80x87 stack is not full, the floating-point argument is assigned floating-point register ST(0) (the top element of the 80x87 stack). The previous top element (if there was one) is now in ST(1). Since arguments are pushed on the stack from right to left, the leftmost floating-point argument will be in ST(0). Otherwise the floating-point argument is assigned a position on the 80x86 stack.

Consider the following example.

```
#pragma aux myrtn parm [8087];

void main()
{
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77;
    j = 77;
    myrtn( x, i, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument of type **float** (4 bytes), the second argument is of type **int** (2 bytes), the third argument is of type **double** (8 bytes) and the fourth argument is of type **long int** (4 bytes). These arguments will be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float,** will be passed in an 80x87 floating-point register.

2. The second argument will be passed on the stack since no 80x86 registers were specified in the register set.

3. The third argument will also be passed on the stack. Remember the following rule: once an argument is assigned a position on the stack, all remaining arguments will be assigned a position on the stack. Note that the above rule holds even though there are some 80x87 floating-point registers available for passing floating-point arguments.

4. The fourth argument will also be passed on the stack.

Let us change the auxiliary pragma in the above example as follows.

```
#pragma aux myrtn parm [ax 8087];
```

The arguments will now be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float** will be passed in an 80x87 floating-point register.

2. The second argument will be passed in register AX, exhausting the set of available 80x86 registers for argument passing.

3. The third argument, being of type **double**, will also be passed in an 80x87 floating-point register.

4. The fourth argument will be passed on the stack since no 80x86 registers remain in the register set.

8.25.12.2 Using the 80x87 to Return Function Values

The following form of the auxiliary pragma can be used to describe a function that returns a floating-point value in ST(0).

```
#pragma aux sym value reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set containing the string "8087", i.e. [8087].

8.25.12.3 Preserving 80x87 Floating-Point Registers Across Calls

The code generator assumes that all eight 80x87 floating-point registers are available for use within a function unless the "fpr" option is used to generate backward compatible code (older Watcom compilers used four registers as a cache). The following form of the auxiliary pragma specifies that the floating-point registers in the 80x87 cache may be modified by the specified function.

```
#pragma aux sym modify reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set containing the string "8087", i.e. [8087].

This instructs Watcom C/C++ to save any local variables that are located in the 80x87 cache before calling the specified routine.

32-bit Topics

9 32-bit Memory Models

9.1 Introduction

This chapter describes the various 32-bit memory models supported by Watcom C/C++. Each memory model is distinguished by two properties; the code model used to implement function calls and the data model used to reference data.

9.2 32-bit Code Models

There are two code models;

1. the small code model and
2. the big code model.

A small code model is one in which all calls to functions are made with *near calls.* In a near call, the destination address is 32 bits and is relative to the segment value in segment register CS. Hence, in a small code model, all code comprising your program, including library functions, must be less than 4GB.

A big code model is one in which all calls to functions are made with *far calls.* In a far call, the destination address is 48 bits (a 16-bit segment value and a 32-bit offset relative to the segment value). This model allows the size of the code comprising your program to exceed 4GB.

> ***Note:*** If your program contains less than 4GB of code, you should use a memory model that employs the small code model. This will result in smaller and faster code since near calls are smaller instructions and are processed faster by the CPU.

9.3 32-bit Data Models

There are two data models;

1. the small data model and
2. the big data model.

A small data model is one in which all references to data are made with *near pointers.* Near pointers are 32 bits; all data references are made relative to the segment value in segment register DS. Hence, in a small data model, all data comprising your program must be less than 4GB.

A big data model is one in which all references to data are made with *far pointers.* Far pointers are 48 bits (a 16-bit segment value and a 32-bit offset relative to the segment value). This removes the 4GB limitation on data size imposed by the small data model. However, when a far pointer is incremented, only the offset is adjusted. Watcom C/C++ assumes that the offset portion of a far pointer will not be incremented beyond 4GB. The compiler will assign an object to a new segment if the grouping of data in a segment will cause the object to cross a segment boundary. Implicit in this is the requirement that no individual object exceed 4GB.

> ***Note:*** If your program contains less than 4GB of data, you should use the small data model. This will result in smaller and faster code since references using near pointers produce fewer instructions.

9.4 Summary of 32-bit Memory Models

As previously mentioned, a memory model is a combination of a code model and a data model. The following table describes the memory models supported by Watcom C/C++.

Memory Model	Code Model	Data Model	Default Code Pointer	Default Data Pointer
flat	small	small	near	near
small	small	small	near	near
medium	big	small	far	near
compact	small	big	near	far
large	big	big	far	far

9.5 Flat Memory Model

In the flat memory model, the application's code and data must total less than 4GB in size. Segment registers CS, DS, SS and ES point to the same linear address space (this does not imply that the segment registers contain the same value). That is, a given offset in one segment refers to the same memory location as that offset in another segment. Essentially, a flat model operates as if there were no segments.

9.6 Mixed 32-bit Memory Model

A mixed memory model application combines elements from the various code and data models. A mixed memory model application might be characterized as one that uses the ***near, far,*** or ***huge*** keywords when describing some of its functions or data objects.

For example, a medium memory model application that uses some far pointers to data can be described as a mixed memory model. In an application such as this, most of the data is in a 4GB segment (DGROUP) and hence can be referenced with near pointers relative to the segment value in segment register DS. This results in more efficient code being generated and better execution times than one can expect from a big data model. Data objects outside of the DGROUP segment are described with the ***far*** keyword.

9.7 Linking Applications for the Various 32-bit Memory Models

Each memory model requires different run-time and floating-point libraries. Each library assumes a particular memory model and should be linked only with modules that have been compiled with the same memory model. The following table lists the libraries that are to be used to link an application that has been compiled for a particular memory model. Currently, only libraries for the flat/small memory model are provided.

Memory Model	Run-time Library	Floating-Point Library (80x87)	Floating-Point Library (f-p calls)
flat/small	CLIB3R.LIB	MATH387R.LIB	MATH3R.LIB
	CLIB3S.LIB	MATH387S.LIB	MATH3S.LIB
	PLIB3R.LIB	CPLX73R.LIB	CPLX3R.LIB
	PLIB3S.LIB	CPLX73S.LIB	CPLX3S.LIB

The letter "R" or "S" which is affixed to the file name indicates the particular strategy with which the modules in the library have been compiled.

R denotes a version of the Watcom C/C++ 32-bit libraries which have been compiled for the "flat/small" memory models using the "3r", "4r" or "5r" option.

S denotes a version of the Watcom C/C++ 32-bit libraries which have been compiled for the "flat/small" memory models using the "3s", "4s" or "5s" option.

9.8 Memory Layout

The following describes the segment ordering of an application linked by the Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all "USE16" segments. These segments are present in applications that execute in both real mode and protected mode. They are first in the segment ordering so that the "REALBREAK" option of the "RUNTIME" directive can be used to separate the real-mode part of the application from the protected-mode part of the application. Currently, the "RUNTIME" directive is valid for Phar Lap executables only.

2. all segments not belonging to group "DGROUP" with class "CODE"

3. all other segments not belonging to group "DGROUP"

4. all segments belonging to group "DGROUP" with class "BEGDATA"

5. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

6. all segments belonging to group "DGROUP" with class "BSS"

7. all segments belonging to group "DGROUP" with class "STACK"

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Watcom C/C++.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<module>_TEXT" where <module> is the file name of the source file.

2. The "FAR_DATA" class consists of the following:

   (a) data objects whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (b) data objects defined using the "FAR" or "HUGE" keyword,

   (c) literals whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (d) literals defined using the "FAR" or "HUGE" keyword.

You can override the default naming convention used by Watcom C/C++ to name segments.

1. The Watcom C/C++ "nm" option can be used to change the name of the module. This, in turn, changes the name of the code segment when compiling for a big code model.
2. The Watcom C/C++ "nt" option can be used to specify the name of the code segment regardless of the code model used.

10 32-bit Assembly Language Considerations

10.1 Introduction

This chapter will deal with the following topics.

1. The data representation of the basic types supported by Watcom C/C++.
2. The memory layout of a Watcom C/C++ program.
3. The method for passing arguments and returning values.
4. The two methods for passing floating-point arguments and returning floating-point values.

   One method is used when one of the Watcom C/C++ "fpi" or "fpi87" options is specified for the generation of in-line 80x87 instructions. When the "fpi" option is specified, an 80x87 emulator is included from a math library if the application includes floating-point operations. When the "fpi87" option is used exclusively, the 80x87 emulator will not be included.

   The other method is used when the Watcom C/C++ "fpc" option is specified. In this case, the compiler generates calls to floating-point support routines in the alternate math libraries.

An understanding of the Intel 80x86 architecture is assumed.

10.2 Data Representation

This section describes the internal or machine representation of the basic types supported by Watcom C/C++.

10.2.1 Type "char"

An item of type "char" occupies 1 byte of storage. Its value is in the following range.

```
0 <= n <= 255
```

Note that "char" is, by default, unsigned. The Watcom C/C++ compiler option "j" can be used to change the default from unsigned to signed. If "char" is signed, an item of type "char" is in the following range.

```
-128 <= n <= 127
```

You can force an item of type "char" to be unsigned or signed regardless of the default by defining them to be of type "unsigned char" or "signed char" respectively.

10.2.2 Type "short int"

An item of type "short int" occupies 2 bytes of storage. Its value is in the following range.

```
-32768 <= n <= 32767
```

Note that "short int" is signed and hence "short int" and "signed short int" are equivalent. If an item of type "short int" is to be unsigned, it must be defined as "unsigned short int". In this case, its value is in the following range.

```
0 <= n <= 65535
```

10.2.3 Type "long int"

An item of type "long int" occupies 4 bytes of storage. Its value is in the following range.

```
-2147483648 <= n <= 2147483647
```

Note that "long int" is signed and hence "long int" and "signed long int" are equivalent. If an item of type "long int" is to be unsigned, it must be defined as "unsigned long int". In this case, its value is in the following range.

```
0 <= n <= 4294967295
```

10.2.4 Type "int"

An item of type "int" occupies 4 bytes of storage. Its value is in the following range.

```
-2147483648 <= n <= 2147483647
```

Note that "int" is signed and hence "int" and "signed int" are equivalent. If an item of type "int" is to be unsigned, it must be defined as "unsigned int". In this case its value is in the following range.

```
0 <= n <= 4294967295
```

If you are generating code that executes in 32-bit mode, "long int" and "int" are equivalent, "unsigned long int" and "unsigned int" are equivalent, and "signed long int" and "signed int" are equivalent. This may not be the case in other environments where "int" and "short int" are 2 bytes.

10.2.5 Type "float"

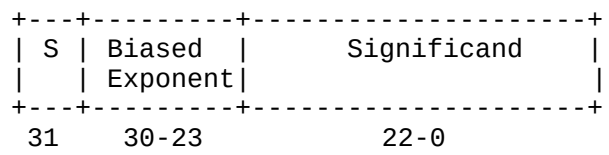
A datum of type "float" is an approximate representation of a real number. Each datum of type "float" occupies 4 bytes. If `m` is the magnitude of `x` (an item of type "float") then `x` can be approximated if

$$2^{-126} <= m < 2^{128}$$

or in more approximate terms if

```
1.175494e-38 <= m <= 3.402823e38
```

Data of type "float" are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.



Notes

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 127 (i.e., exponent value 1 represents 2^{-126}; exponent value 127 represents 2^{0}; exponent value 254 represents 2^{127}; etc.). The exponent field is 8 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 23 bits long.

Zero A real zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

10.2.6 Type "double"

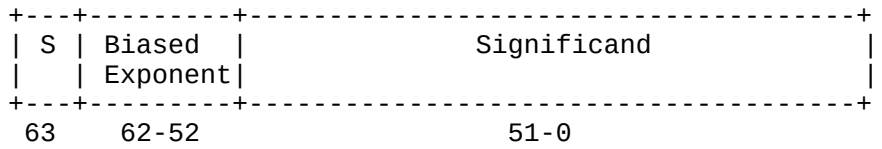
A datum of type "double" is an approximate representation of a real number. The precision of a datum of type "double" is greater than or equal to one of type "float". Each datum of type "double" occupies 8 bytes. If `m` is the magnitude of `x` (an item of type "double") then `x` can be approximated if

$$2^{-1022} <= m < 2^{1024}$$

or in more approximate terms if

```
2.2250738585072e-308 <= m <= 1.79769313486232e308
```

Data of type "double" are represented internally as follows. Note that bytes are stored in memory with the least significant byte first and the most significant byte last.



Notes:

S S = Sign bit (0=positive, 1=negative)

Exponent The exponent bias is 1023 (i.e., exponent value 1 represents 2^{-1022}; exponent value 1023 represents 2^{0}; exponent value 2046 represents 2^{1023}; etc.). The exponent field is 11 bits long.

Significand The leading bit of the significand is always 1, hence it is not stored in the significand field. Thus the significand is always "normalized". The significand field is 52 bits long.

Zero A double precision zero quantity occurs when the sign bit, exponent, and significand are all zero.

Infinity When the exponent field is all 1 bits and the significand field is all zero bits then the quantity represents positive or negative infinity, depending on the sign bit.

Not Numbers When the exponent field is all 1 bits and the significand field is non-zero then the quantity is a special value called a NAN (Not-A-Number).

When the exponent field is all 0 bits and the significand field is non-zero then the quantity is a special value called a "denormal" or nonnormal number.

10.3 Memory Layout

The following describes the segment ordering of an application linked by the Watcom Linker. Note that this assumes that the "DOSSEG" linker option has been specified.

1. all "USE16" segments. These segments are present in applications that execute in both real mode and protected mode. They are first in the segment ordering so that the "REALBREAK" option of the "RUNTIME" directive can be used to separate the real-mode part of the application from the protected-mode part of the application. Currently, the "RUNTIME" directive is valid for Phar Lap executables only.

2. all segments not belonging to group "DGROUP" with class "CODE"

3. all other segments not belonging to group "DGROUP"

4. all segments belonging to group "DGROUP" with class "BEGDATA"

5. all segments belonging to group "DGROUP" not with class "BEGDATA", "BSS" or "STACK"

6. all segments belonging to group "DGROUP" with class "BSS"

7. all segments belonging to group "DGROUP" with class "STACK"

Segments belonging to class "BSS" contain uninitialized data. Note that this only includes uninitialized data in segments belonging to group "DGROUP". Segments belonging to class "STACK" are used to define the size of the stack used for your application. Segments belonging to the classes "BSS" and "STACK" are last in the segment ordering so that uninitialized data need not take space in the executable file.

In addition to these special segments, the following conventions are used by Watcom C/C++.

1. The "CODE" class contains the executable code for your application. In a small code model, this consists of the segment "_TEXT". In a big code model, this consists of the segments "<module>_TEXT" where <module> is the file name of the source file.

2. The "FAR_DATA" class consists of the following:

   (a) data objects whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (b) data objects defined using the "FAR" or "HUGE" keyword,

   (c) literals whose size exceeds the data threshold in large data memory models (the data threshold is 32K unless changed using the "zt" compiler option)

   (d) literals defined using the "FAR" or "HUGE" keyword.

You can override the default naming convention used by Watcom C/C++ to name segments.

1. The Watcom C/C++ "nm" option can be used to change the name of the module. This, in turn, changes the name of the code segment when compiling for a big code model.
2. The Watcom C/C++ "nt" option can be used to specify the name of the code segment regardless of the code model used.

10.4 Calling Conventions for Non-80x87 Applications

The following sections describe the calling convention used when compiling with the "fpc" compiler option.

10.4.1 Passing Arguments Using Register-Based Calling Conventions

How arguments are passed to a function with register-based calling conventions is determined by the size (in bytes) of the argument and where in the argument list the argument appears. Depending on the size, arguments are either passed in registers or on the stack. Arguments such as structures are almost always passed on the stack since they are generally too large to fit in registers. Since arguments are processed from left to right, the first few arguments are likely to be passed in registers (if they can fit) and, if the argument list contains many arguments, the last few arguments are likely to be passed on the stack.

The registers used to pass arguments to a function are EAX, EBX, ECX and EDX. The following algorithm describes how arguments are passed to functions.

Initially, we have the following registers available for passing arguments: EAX, EDX, EBX and ECX. Note that registers are selected from this list in the order they appear. That is, the first register selected is EAX and the last is ECX. For each argument *Ai*, starting with the left most argument, perform the following steps.

1. If the size of *Ai* is 1 byte or 2 bytes, convert it to 4 bytes and proceed to the next step. If *Ai* is of type "unsigned char" or "unsigned short int", it is converted to an "unsigned int". If *Ai* is of type "signed char" or "signed short int", it is converted to a "signed int". If *Ai* is a 1-byte or 2-byte structure, the padding is determined by the compiler.

2. If an argument has already been assigned a position on the stack, *Ai* will also be assigned a position on the stack. Otherwise, proceed to the next step.

3. If the size of *Ai* is 4 bytes, select a register from the list of available registers. If a register is available, *Ai* is assigned that register. The register is then removed from

the list of available registers. If no registers are available, *Ai* will be assigned a position on the stack.

4. If the type of *Ai* is "far pointer", select a register pair from the following list of combinations: [EDX EAX] or [ECX EBX]. The first available register pair is assigned to *Ai* and removed from the list of available pairs. The segment value will actually be passed in register DX or CX and the offset in register EAX or EBX. If none of the above register pairs is available, *Ai* will be assigned a position on the stack. Note that 8 bytes will be pushed on the stack even though the size of an item of type "far pointer" is 6 bytes.

5. If the type of *Ai* is "double" or "float" (in the absence of a function prototype), select a register pair from the following list of combinations: [EDX EAX] or [ECX EBX]. The first available register pair is assigned to *Ai* and removed from the list of available pairs. The high-order 32 bits of the argument are assigned to the first register in the pair; the low-order 32 bits are assigned to the second register in the pair. If none of the above register pairs is available, *Ai* will be assigned a position on the stack.

6. All other arguments will be assigned a position on the stack.

Notes:

1. Arguments that are assigned a position on the stack are padded to a multiple of 4 bytes. That is, if a 3-byte structure is assigned a position on the stack, 4 bytes will be pushed on the stack.

2. Arguments that are assigned a position on the stack are pushed onto the stack starting with the rightmost argument.

10.4.2 Sizes of Predefined Types

The following table lists the predefined types, their size as returned by the "sizeof" function, the size of an argument of that type and the registers used to pass that argument if it was the only argument in the argument list.

Basic Type	*"sizeof"*	*Argument Size*	*Registers Used*
char	1	4	[EAX]
short int	2	4	[EAX]
int	4	4	[EAX]
long int	4	4	[EAX]
float	4	8	[EDX EAX]

double	8	8	[EDX EAX]
near pointer	4	4	[EAX]
far pointer	6	8	[EDX EAX]

Note that the size of the argument listed in the table assumes that no function prototypes are specified. Function prototypes affect the way arguments are passed. This will be discussed in the section entitled "Effect of Function Prototypes on Arguments".

Notes:

1. Provided no function prototypes exist, an argument will be converted to a default type as described in the following table.

Argument Type	***Passed As***
char	unsigned int
signed char	signed int
unsigned char	unsigned int
short	unsigned int
signed short	signed int
unsigned short	unsigned int
float	double

10.4.3 Size of Enumerated Types

The integral type of an enumerated type is determined by the values of the enumeration constants. In strict ANSI C mode, all enumerated constants are of type `int`. In the extensions mode, the compiler will use the smallest integral type possible (excluding `long` ints) that can represent all values of the enumerated type. For instance, if the minimum and maximum values of the enumeration constants are in the range −128 and 127, the enumerated type will be equivalent to a `signed char` (size = 1 byte). All references to enumerated constants in the previous instance will have type `signed char`. An enumerated constant is always promoted to an `int` when passed as an argument.

10.4.4 Effect of Function Prototypes on Arguments

Function prototypes define the types of the formal parameters of a function. Their appearance affects the way in which arguments are passed. An argument will be converted to the type of the corresponding formal parameter in the function prototype. Consider the following example.

```
void prototype( float x, int i );

void main()
{
  float x;
  int   i;

  x = 3.14;
  i = 314;
  prototype( x, i );
  rtn( x, i );
}
```

The function prototype for `prototype` specifies that the first argument is to be passed as a "float" and the second argument is to be passed as an "int". This results in the first argument being passed in register EAX and the second argument being passed in register EDX.

If no function prototype is given, as is the case for the function `rtn,` the first argument will be passed as a "double" and the second argument would be passed as an "int". This results in the first argument being passed in registers EDX and EAX and the second argument being passed in register EBX.

Note that even though both `prototype` and `rtn` were called with identical argument lists, the way in which the arguments were passed was completely different simply because a function prototype for `prototype` was specified. Function prototyping is an excellent way to guarantee that arguments will be passed as expected to your assembly language function.

10.4.5 Interfacing to Assembly Language Functions

Consider the following example.

Example:

```
void main()
{
    double   x;
    int      i;
    double   y;

    x = 7;
    i = 77;
    y = 777;
    myrtn( x, i, y );
}
```

`myrtn` is an assembly language function that requires three arguments. The first argument is of type "double", the second argument is of type "int" and the third argument is again of type "double". Using the rules for register-based calling conventions, these arguments will be passed to `myrtn` in the following way:

1. The first argument will be passed in registers EDX and EAX leaving EBX and ECX as available registers for other arguments.

2. The second argument will be passed in register EBX leaving ECX as an available register for other arguments.

3. The third argument will not fit in register ECX (its size is 8 bytes) and hence will be pushed on the stack.

Let us look at the stack upon entry to `myrtn.`

Small Code Model

```
Offset
        +-----------------+
  0     | return address | <- ESP points here
        +-----------------+
  4     | argument #3    |
        |                 |
        +-----------------+
 12     |                 |
```

Big Code Model

```
Offset
        +-----------------+
  0     | return address | <- ESP points here
        |                 |
        +-----------------+
  8     | argument #3    |
        |                 |
        +-----------------+
 16     |                 |
```

Notes:

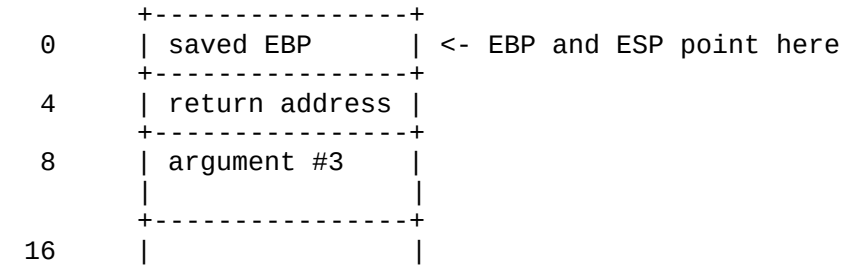
1. The return address is the top element on the stack. In a small code model, the return address is 1 double word (32 bits)

Register EBP is normally used to address arguments on the stack. Upon entry to the function, register EBP is set to point to the stack but before doing so we must save its contents. The following two instructions achieve this.

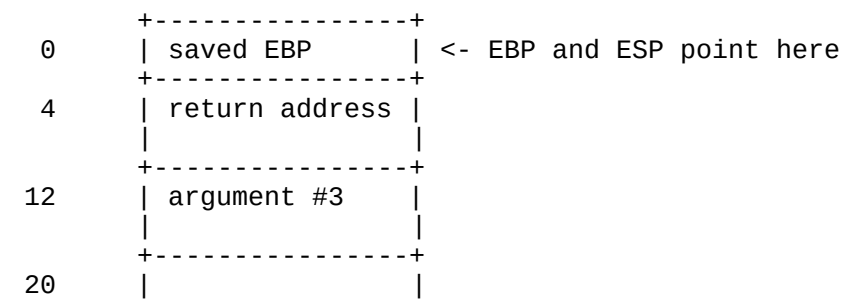
```
push    EBP             ; save current value of EBP
mov     EBP,ESP         ; get access to arguments
```

After executing these instructions, the stack looks like this.

Small Code Model



Big Code Model



As the above diagrams show, the third argument is at offset 8 from register EBP in a small code model and offset 12 in a big code model.

Upon exit from `myrtn`, we must restore the value of EBP. The following two instructions achieve this.

```
mov     ESP,EBP         ; restore stack pointer
pop     EBP             ; restore EBP
```

The following is a sample assembly language function which implements `myrtn`.

Small Memory Model (small code, small data)

```
DGROUP     group     _DATA, _BSS
_TEXT      segment byte public 'CODE'
           assume    CS:_TEXT
           assume    DS:DGROUP
           public    myrtn_
myrtn_     proc      near
           push      EBP          ; save EBP
           mov       EBP,ESP      ; get access to arguments
;
; body of function
;
           mov       ESP,EBP      ; restore ESP
           pop       EBP          ; restore EBP
           ret       8            ; return and pop last arg
myrtn_     endp
_TEXT      ends
```

Large Memory Model (big code, big data)

```
DGROUP     group     _DATA, _BSS
MYRTN_TEXT segment byte public 'CODE'
           assume    CS:MYRTN_TEXT
           public    myrtn_
myrtn_     proc      far
           push      EBP          ; save EBP
           mov       EBP,ESP      ; get access to arguments
;
; body of function
;
           mov       ESP,EBP      ; restore ESP
           pop       EBP          ; restore EBP
           ret       8            ; return and pop last arg
myrtn_     endp
MYRTN_TEXT ends
```

Notes:

1. Global function names must be followed with an underscore. Global variable names must be preceded with an underscore.

2. All used 80x86 registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to saved and restored if you are compiling your application with the "r" option.

3. The direction flag must be clear before returning to the caller.

4. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.

5. In a small data model, segment register DS contains the segment address of the group "DGROUP". This is not the case in a big data model.

6. When writing assembly language functions for the small code model, you must declare them as "near". If you wish to write assembly language functions for the big code model, you must declare them as "far".

7. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

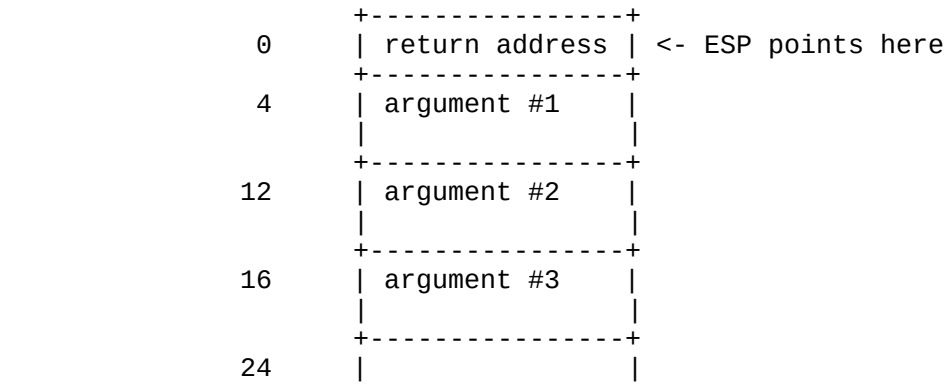
8. If any of the arguments was pushed onto the stack, the called routine must pop those arguments off the stack in the "ret" instruction.

10.4.6 Using Stack-Based Calling Conventions

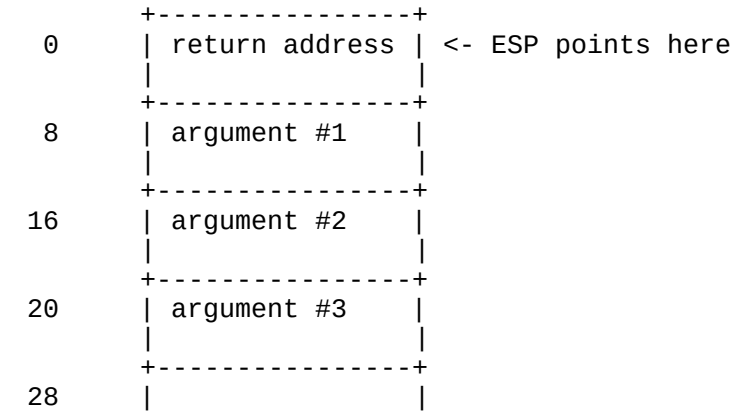
Let us now consider the example in the previous section except this time we will use the stack-based calling convention. The most significant difference between the stack-based calling convention and the register-based calling convention is the way the arguments are passed. When using the stack-based calling conventions, no registers are used to pass arguments. Instead, all arguments are passed on the stack.

Let us look at the stack upon entry to `myrtn`.

Small Code Model



Big Code Model



Notes:

1. The return address is the top element on the stack. In a small code model, the return address is 1 double word (32 bits)

Register EBP is normally used to address arguments on the stack. Upon entry to the function, register EBP is set to point to the stack but before doing so we must save its contents. The following two instructions achieve this.

```
push    EBP             ; save current value of EBP
mov     EBP,ESP         ; get access to arguments
```

After executing these instructions, the stack looks like this.

Small Code Model

```
Offset
        +----------------+
  0     | saved EBP      | <- EBP and ESP point here
        +----------------+
  4     | return address |
        +----------------+
  8     | argument #1    |
        |                |
        +----------------+
 16     | argument #2    |
        |                |
        +----------------+
 20     | argument #3    |
        |                |
        +----------------+
 28     |                |
```

Big Code Model

```
Offset
        +----------------+
  0     | saved EBP      | <- EBP and ESP point here
        +----------------+
  4     | return address |
        |                |
        +----------------+
 12     | argument #1    |
        |                |
        +----------------+
 20     | argument #2    |
        |                |
        +----------------+
 24     | argument #3    |
        |                |
        +----------------+
 32     |                |
```

As the above diagrams show, the argument are all on the stack and are referenced by specifying an offset from register EBP.

Upon exit from `myrtn`, we must restore the value of EBP. The following two instructions achieve this.

```
mov     ESP,EBP       ; restore stack pointer
pop     EBP           ; restore EBP
```

The following is a sample assembly language function which implements `myrtn`.

Small Memory Model (small code, small data)

```
DGROUP     group     _DATA, _BSS
_TEXT      segment byte public 'CODE'
           assume    CS:_TEXT
           assume    DS:DGROUP
           public    myrtn
myrtn      proc      near
           push      EBP          ; save EBP
           mov       EBP,ESP      ; get access to arguments
;
; body of function
;
           mov       ESP,EBP      ; restore ESP
           pop       EBP          ; restore EBP
           ret
           ; return
myrtn      endp
_TEXT      ends
```

Large Memory Model (big code, big data)

```
DGROUP     group     _DATA, _BSS
MYRTN_TEXT segment byte public 'CODE'
           assume    CS:MYRTN_TEXT
           public    myrtn
myrtn      proc      far
           push      EBP          ; save EBP
           mov       EBP,ESP      ; get access to arguments
;
; body of function
;
           mov       ESP,EBP      ; restore ESP
           pop       EBP          ; restore EBP
           ret
           ; return
myrtn      endp
MYRTN_TEXT ends
```

Notes:

1. Global function names must not be followed with an underscore as was the case with the register-based calling convention. Global variable names must not be preceded with an underscore as was the case with the register-based calling convention.

2. All used 80x86 registers except registers EAX, ECX and EDX must be saved on entry and restored on exit. Segment registers DS and ES must also be saved on entry and restored on exit. Segment register ES does not have to be saved and restored when using a memory model that is not a small data model. Note that segment registers only have to be saved and restored if you are compiling your application with the "r" option.

3. The direction flag must be clear before returning to the caller.

4. In a small code model, any segment containing executable code must belong to the segment "_TEXT" and the class "CODE". The segment "_TEXT" must have a "combine" type of "PUBLIC". On entry, CS contains the segment address of the segment "_TEXT". In a big code model there is no restriction on the naming of segments which contain executable code.

5. In a small data model, segment register DS contains the segment address of the group "DGROUP". This is not the case in a big data model.

6. When writing assembly language functions for the small code model, you must declare them as "near". If you wish to write assembly language functions for the big code model, you must declare them as "far".

7. In general, when naming segments for your code or data, you should follow the conventions described in the section entitled "Memory Layout" in this chapter.

8. The caller is responsible for removing arguments from the stack.

10.4.7 Functions with Variable Number of Arguments

A function prototype with a parameter list that ends with ",..." has a variable number of arguments. In this case, all arguments are passed on the stack. Since no prototyping information exists for arguments represented by ",...", those arguments are passed as described in the section "Passing Arguments".

10.4.8 Returning Values from Functions

The way in which function values are returned depends on the size of the return value. The following examples describe how function values are to be returned. They are coded for a small code model.

1. 1-byte values are to be returned in register AL.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret1_
Ret1_   proc    near   ; char Ret1()
        mov     AL,'G'
        ret
Ret1_   endp
_TEXT   ends
        end
```

2. 2-byte values are to be returned in register AX.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret2_
Ret2_   proc    near   ; short int Ret2()
        mov     AX,77
        ret
Ret2_   endp
_TEXT   ends
        end
```

3. 4-byte values are to be returned in register EAX.

Example:

```
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  Ret4_
Ret4_   proc    near   ; int Ret4()
        mov     EAX,7777777
        ret
Ret4_   endp
_TEXT   ends
        end
```

4. 8-byte values, except structures, are to be returned in registers EDX and EAX. When using the "fpc" (floating-point calls) option, "float" and "double" are returned in registers. See section "Returning Values in 80x87-based Applications" when using the "fpi" or "fpi87" options.

Example:

```
            .8087
    _TEXT   segment byte public 'CODE'
            assume  CS:_TEXT
            public  Ret8_
    Ret8_   proc    near   ; double Ret8()
            mov     EDX,dword ptr CS:Val8+4
            mov     EAX,dword ptr CS:Val8
            ret
    Val8:   dq      7.7
    Ret8_   endp
    _TEXT   ends
            end
```

The ".8087" pseudo-op must be specified so that all floating-point constants are generated in 8087 format.

5. Otherwise, the caller allocates space on the stack for the return value and sets register ESI to point to this area. In a big data model, register ESI contains an offset relative to the segment value in segment register SS.

   Example:

```
    _TEXT   segment byte public 'CODE'
            assume  CS:_TEXT
            public  RetX_
    ;
    ; struct int_values {
    ;     int value1, value2, value3, value4, value5;
    ;                   };
    ;
    RetX_   proc    near ; struct int_values RetX()
            mov     dword ptr SS:0[ESI],71
            mov     dword ptr SS:4[ESI],72
            mov     dword ptr SS:8[ESI],73
            mov     dword ptr SS:12[ESI],74
            mov     dword ptr SS:16[ESI],75
            ret
    RetX_   endp
    _TEXT   ends
            end
```

   When returning values on the stack, remember to use a segment override to the stack segment (SS).

The following is an example of a Watcom C/C++ program calling the above assembly language subprograms.

```
#include <stdio.h>

struct int_values {
    int value1;
    int value2;
    int value3;
    int value4;
    int value5;
};

extern  char              Ret1(void);
extern  short int         Ret2(void);
extern  long int          Ret4(void);
extern  double            Ret8(void);
extern  struct int_values RetX(void);

void main()
{
    struct int_values x;

    printf( "Ret1 = %c\n", Ret1() );
    printf( "Ret2 = %d\n", Ret2() );
    printf( "Ret4 = %ld\n", Ret4() );
    printf( "Ret8 = %f\n", Ret8() );
    x = RetX();
    printf( "RetX1 = %d\n", x.value1 );
    printf( "RetX2 = %d\n", x.value2 );
    printf( "RetX3 = %d\n", x.value3 );
    printf( "RetX4 = %d\n", x.value4 );
    printf( "RetX5 = %d\n", x.value5 );
}
```

The above function should be compiled for a small code model (use the "mf", "ms" or "mc" compiler option).

> ***Note:*** Returning values from functions in the stack-based calling convention is the same as returning values from functions in the register-based calling convention when using the "fpc" option.

10.5 Calling Conventions for 80x87-based Applications

When a source file is compiled by Watcom C/C++ with one of the "fpi" or "fpi87" options, all floating-point arguments are passed on the 80x86 stack. The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, it is assigned a position on the 80x86 stack.

> ***Note:*** When compiling using the "fpi" or "fpi87" options, the method used for passing floating-point arguments in the stack-based calling convention is identical to the method used in the register-based calling convention. However, when compiling using the "fpi" or "fpi87" options, the method used for returning floating-point values in the stack-based calling convention is different from the method used in the register-based calling convention. The register-based calling convention returns floating-point values in ST(0), whereas the stack-based calling convention returns floating-point values in EDX and EAX.

10.5.1 Passing Values in 80x87-based Applications

Consider the following example.

Example:

```
extern  void    myrtn(int,float,double,long int);

void main()
{
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77
    j = 77;
    myrtn( i, x, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument is of type "int" (4 bytes), the second argument is of type "float" (4 bytes), the third argument is of type "double" (8 bytes) and the fourth argument is of type "long int" (4 bytes).

When using the stack-based calling conventions, all of the arguments will be passed on the stack. When using the register-based calling conventions, the above arguments will be passed to `myrtn` in the following way:

1. The first argument will be passed in register EAX leaving EBX, ECX and EDX as available registers for other arguments.

2. The second argument will be passed on the 80x86 stack since it is a floating-point argument.

3. The third argument will also be passed on the 80x86 stack since it is a floating-point argument.

4. The fourth argument will be passed on the 80x86 stack since a previous argument has been assigned a position on the 80x86 stack.

Remember, arguments are pushed on the stack from right to left. That is, the rightmost argument is pushed first.

Any assembly language function must obey the following rule.

1. All arguments passed on the stack must be removed by the called function.

The following is a sample assembly language function which implements `myrtn.`

Example:

```
        .8087
_TEXT   segment byte public 'CODE'
        assume  CS:_TEXT
        public  myrtn_
myrtn_  proc    near
;
; body of function
;
        ret 16          ; return and pop arguments
myrtn_  endp
_TEXT   ends
        end
```

Notes:

1. Function names must be followed by an underscore.

2. All used 80x86 registers must be saved on entry and restored on exit except those used to pass arguments and return values. Note that segment registers only have to saved and restored if you are compiling your application with the "r" option. In this example, EAX does not have to be saved as it was used to pass the first argument. Floating-point registers can be modified without saving their contents.

3. The direction flag must be clear before returning to the caller.

4. This function has been written for a small code model. Any segment containing executable code must belong to the class "CODE" and the segment "_TEXT". On entry, CS contains the segment address of the segment "_TEXT". The above restrictions do not apply in a big code memory model.

5. When writing assembly language functions for a small code model, you must declare them as "near". If you wish to write assembly language functions for a big code model, you must declare them as "far".

10.5.2 Returning Values in 80x87-based Applications

When using the stack-based calling conventions with "fpi" or "fpi87", floating-point values are returned in registers. Single precision values are returned in EAX, and double precision values are returned in EDX:EAX.

When using the register-based calling conventions with "fpi" or "fpi87", floating-point values are returned in ST(0). All other values are returned in the manner described earlier in this chapter.

11 32-bit Pragmas

11.1 Introduction

A pragma is a compiler directive that provides the following capabilities.

- Pragmas allow you to specify certain compiler options.
- Pragmas can be used to direct the Watcom C/C++ code generator to emit specialized sequences of code for calling functions which use argument passing and value return techniques that differ from the default used by Watcom C/C++.
- Pragmas can be used to describe attributes of functions (such as side effects) that are not possible at the C/C++ language level. The code generator can use this information to generate more efficient code.
- Any sequence of in-line machine language instructions, including DOS and BIOS function calls, can be generated in the object code.

Pragmas are specified in the source file using the ***pragma*** directive. The following notation is used to describe the syntax of pragmas.

keywords A `keyword` is shown in a mono-spaced courier font.

program-item A ***program-item*** is shown in a roman bold-italics font. A ***program-item*** is a symbol name or numeric value supplied by the programmer.

punctuation A `punctuation character` shown in a mono-spaced courier font must be entered as is.

A ***punctuation character*** shown in a roman bold-italics font is used to describe syntax. The following syntactical notation is used.

[abc] The item *abc* is optional.

{abc} The item *abc* may be repeated zero or more times.

a|b|c One of *a*, *b* or *c* may be specified.

a ::= b The item *a* is defined in terms of *b*.

(a) Item *a* is evaluated first.

The following classes of pragmas are supported.

- pragmas that specify options
- pragmas that specify default libraries
- pragmas that describe the way structures are stored in memory
- pragmas that provide auxiliary information used for code generation

11.2 Using Pragmas to Specify Options

Currently, the following options can be specified with pragmas:

unreferenced The "unreferenced" option controls the way Watcom C/C++ handles unused symbols. For example,

```
#pragma on (unreferenced);
```

will cause Watcom C/C++ to issue warning messages for all unused symbols. This is the default. Specifying

```
#pragma off (unreferenced);
```

will cause Watcom C/C++ to ignore unused symbols. Note that if the warning level is not high enough, warning messages for unused symbols will not be issued even if "unreferenced" was specified.

check_stack The "check_stack" option controls the way stack overflows are to be handled. For example,

```
#pragma on (check_stack);
```

will cause stack overflows to be detected and

```
#pragma off (check_stack);
```

will cause stack overflows to be ignored. When "check_stack" is on, Watcom C/C++ will generate a run-time call to a stack-checking routine at the start of every routine compiled. This run-time routine will issue an error if a stack overflow occurs when invoking the routine. The default is to check for stack overflows. Stack overflow checking is particularly useful when functions are invoked recursively. Note that if the stack overflows and stack checking has been suppressed, unpredictable results can occur.

If a stack overflow does occur during execution and you are sure that your program is not in error (i.e. it is not unnecessarily recursing), you must increase the stack size. This is done by linking your application again and specifying the "STACK" option to the Watcom Linker with a larger stack size.

It is also possible to specify more than one option in a pragma as illustrated by the following example.

```
#pragma on (check_stack unreferenced);
```

reuse_duplicate_strings (C only) (C Only) The "reuse_duplicate_strings" option controls the way Watcom C handles identical strings in an expression. For example,

```
#pragma on (reuse_duplicate_strings);
```

will cause Watcom C to reuse identical strings in an expression. This is the default. Specifying

```
#pragma off (reuse_duplicate_strings);
```

will cause Watcom C to generate additional copies of the identical string. The following example shows where this may be of importance to the way the application behaves.

Example:

```
#include <stdio.h>

#pragma off (reuse_duplicate_strings)


void poke( char *, char * );

void main()
  {
    poke( "Hello world\n", "Hello world\n" );
  }

void poke( char *x, char *y )
  {
    x[3] = 'X';
    printf( x );
    y[4] = 'Y';
    printf( y );
  }
/*
Default output:
HelXo world
HelXY world
*/
```

11.3 Using Pragmas to Specify Default Libraries

Default libraries are specified in special object module records. Library names are extracted from these special records by the Watcom Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched.

By default, that is if no library pragma is specified, the Watcom C/C++ compiler generates, in the object file defining the main program, default libraries corresponding to the memory model and floating-point model used to compile the file. For example, if you have compiled the source file containing the main program for the flat memory model and the floating-point calls floating-point model, the libraries "clib3r" and "math3r" will be placed in the object file.

If you wish to add your own default libraries to this list, you can do so with a library pragma. Consider the following example.

```
#pragma library (mylib);
```

The name "mylib" will be added to the list of default libraries specified in the object file.

If the library specification contains characters such as '\', ':' or ',' (i.e., any character not allowed in a C identifier), you must enclose it in double quotes as in the following example.

```
#pragma library ("\watcom\lib286\dos\graph.lib");
#pragma library ("\watcom\lib386\dos\graph.lib");
```

If you wish to specify more than one library in a library pragma you must separate them with spaces as in the following example.

```
#pragma library (mylib "\watcom\lib286\dos\graph.lib");
#pragma library (mylib "\watcom\lib386\dos\graph.lib");
```

11.4 The ALLOC_TEXT Pragma (C Only)

The "alloc_text" pragma can be used to specify the name of the text segment into which the generated code for a function, or a list of functions, is to be placed. The following describes the form of the "alloc_text" pragma.

```
#pragma alloc_text ( seg_name, fn {, fn} ) [;]
```

where ***description:***

seg_name is the name of the text segment.

fn is the name of a function.

Consider the following example.

```
extern int fn1(int);
extern int fn2(void);
#pragma alloc_text ( my_text, fn1, fn2 );
```

The code for the functions `fn1` and `fn2` will be placed in the segment `my_text`. Note: function prototypes for the named functions must exist prior to the "alloc_text" pragma.

11.5 The CODE_SEG Pragma

The "code_seg" pragma can be used to specify the name of the text segment into which the generated code for functions is to be placed. The following describes the form of the "code_seg" pragma.

```
#pragma code_seg ( seg_name [, class_name] ) [;]
```

where ***description:***

seg_name is the name of the text segment enclosed in quotes. Also, `seg_name` may be a macro as in:

```
#define seg_name "MY_CODE_SEG"
#pragma code_seg ( seg_name );
```

class_name is the optional class name of the text segment enclosed in quotes. Also, `class_name` may be a macro as in:

```
#define class_name "MY_CLASS"
#pragma code_seg ( "MY_CODE_SEG", class_name );
```

Consider the following example.

```
#pragma code_seg ( "my_text" );

int incr( int i )
{
    return( i + 1 );
}

int decr( int i )
{
    return( i - 1 );
}
```

The code for the functions `incr` and `decr` will be placed in the segment `my_text.`

To return to the default segment, do not specify any string between the opening and closing parenthesis.

```
#pragma code_seg ();
```

11.6 The COMMENT Pragma

The "comment" pragma can be used to place a comment record in an object file or executable file. The following describes the form of the "comment" pragma.

```
#pragma comment ( comment_type [, "comment_string"] ) [;]
```

where description:

comment_type specifies the type of comment record. The allowable comment types are:

lib Default libraries are specified in special object module records. Library names are extracted from these special records by the Watcom Linker. When unresolved references remain after processing all object modules specified in linker "FILE" directives, these default libraries are searched after all libraries specified in linker "LIBRARY" directives have been searched.

The "lib" form of this pragma offers the same features as the "library" pragma. See the section entitled "Using Pragmas to Specify Default Libraries" on page 250 for more information.

"comment_string" is an optional string literal that provides additional information for some comment types.

Consider the following example.

```
#pragma comment ( lib, "mylib" );
```

11.7 The DATA_SEG Pragma

The "data_seg" pragma can be used to specify the name of the segment into which data is to be placed. The following describes the form of the "data_seg" pragma.

```
#pragma data_seg ( seg_name [, class_name] ) [;]
```

where ***description:***

seg_name is the name of the data segment enclosed in quotes. Also, `seg_name` may be a macro as in:

```
#define seg_name "MY_DATA_SEG"
#pragma data_seg ( seg_name );
```

class_name is the optional class name of the data segment enclosed in quotes. Also, `class_name` may be a macro as in:

```
#define class_name "MY_CLASS"
#pragma data_seg ( "MY_DATA_SEG", class_name );
```

Consider the following example.

```
#pragma data_seg ( "my_data" );

static int i;
static int j;
```

The data for `i` and `j` will be placed in the segment `my_data`.

To return to the default segment, do not specify any string between the opening and closing parenthesis.

```
#pragma data_seg ();
```

11.8 The DISABLE_MESSAGE Pragma (C Only)

The "disable_message" pragma disables the issuance of specified diagnostic messages. The form of the "disable_message" pragma is as follows.

```
#pragma disable_message ( msg_num {, msg_num} ) [;]
```

where ***description:***

msg_num is the number of the diagnostic message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Watcom C Diagnostic Messages" on page 363. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

See also the description of "The ENABLE_MESSAGE Pragma (C Only)".

11.9 The DUMP_OBJECT_MODEL Pragma (C++ Only)

The "dump_object_model" pragma causes the C++ compiler to print information about the object model for an indicated class or an enumeration name to the diagnostics file. For class names, this information includes the offsets and sizes of fields within the class and within base classes. For enumeration names, this information consists of a list of all the enumeration constants with their values.

The general form of the "dump_object_model" pragma is as follows.

```
#pragma dump_object_model class [;]
#pragma dump_object_model enumeration [;]
class ::= a defined C++ class free of errors
enumeration ::= a defined C++ enumeration name
```

This pragma is designed to be used for information purposes only.

11.10 The ENABLE_MESSAGE Pragma (C Only)

The "enable_message" pragma re-enables the issuance of specified diagnostic messages that have been previously disabled. The form of the "enable_message" pragma is as follows.

```
#pragma enable_message ( msg_num {, msg_num} ) [;]
```

where ***description:***

msg_num is the number of the diagnostic message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Watcom C Diagnostic Messages" on page 363. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

See also the description of "The DISABLE_MESSAGE Pragma (C Only)" on page 254.

11.11 The ENUM Pragma

The "enum" pragma affects the underlying storage-definition for subsequent *enum* declarations. The forms of the "enum" pragma are as follows.

```
#pragma enum int [;]
#pragma enum minimum [;]
#pragma enum original [;]
#pragma enum pop [;]
```

where ***description:***

int Make *int* the underlying storage definition (same as the "ei" compiler option).

minimum Minimize the underlying storage definition (same as not specifying the "ei" compiler option).

original Reset back to the original compiler option setting (i.e., what was or was not specified on the command line).

pop Restore the previous setting.

The first three forms all push the previous setting before establishing the new setting.

11.12 The ERROR Pragma

The "error" pragma can be used to issue an error message with the specified text. The following describes the form of the "error" pragma.

```
#pragma error "error text" [;]
```

where ***description:***

"error text" is the text of the message that you wish to display.

You should use the ANSI ***#error*** directive rather than this pragma. This pragma is provided for compatibility with legacy code. The following is an example.

```
#if defined(__386__)
    ...
#elseif defined(__86__)
    ...
#else
#pragma error ( "neither __386__ or __86__ defined" );
#endif
```

11.13 The EXTREF Pragma

The "extref" pragma is used to generate a reference to an external function or data item. The form of the "extref" pragma is as follows.

```
#pragma extref name [;]
```

where ***description:***

name is the name of an external function or data item. It must be declared to be an external function or data item before the pragma is encountered. In C++, when *name* is a function, it must not be overloaded.

This pragma causes an external reference for the function or data item to be emitted into the object file even if that function or data item is not referenced in the module. The external

reference will cause the linker to include the module containing that name in the linked program or DLL.

This is useful for debugging since you can cause debugging routines (callable from within debugger) to be included into a program or DLL to be debugged.

In C++, you can also force constructors and/or destructors to be called for a data item without necessarily referencing the data item anywhere in your code.

11.14 The FUNCTION Pragma

Certain functions, such as those listed in the description of the "oi" and "om" options, have intrinsic forms. These functions are special functions that are recognized by the compiler and processed in a special way. For example, the compiler may choose to generate in-line code for the function. The intrinsic attribute for these special functions is set by specifying the "oi" or "om" option or using an "intrinsic" pragma. The "function" pragma can be used to remove the intrinsic attribute for a specified list of functions.

The following describes the form of the "function" pragma.

```
#pragma function ( fn {, fn} ) [;]
```

where ***description:***

fn is the name of a function.

Suppose the following source code was compiled using the "om" option so that when one of the special math functions is referenced, the intrinsic form will be used. In our example, we have referenced the function `sin` which does have an intrinsic form. By specifying `sin` in a "function" pragma, the intrinsic attribute will be removed, causing the function `sin` to be treated as a regular user-defined function.

```
#include <math.h>
#pragma function( sin );

double test( double x )
{
    return( sin( x ) );
}
```

11.15 Setting Priority of Static Data Initialization (C++ Only)

The "initialize" pragma sets the priority for initialization of static data in the file. This priority only applies to initialization of static data that requires the execution of code. For example, the initialization of a class that contains a constructor requires the execution of the constructor. Note that if the sequence in which initialization of static data in your program takes place has no dependencies, the "initialize" pragma need not be used.

The general form of the "initialize" pragma is as follows.

```
#pragma initialize [before | after] priority [;]

priority ::= n | library | program
```

where ***description:***

n is a number representing the priority and must be in the range 0-255. The larger the priority, the later the point at which initialization will occur.

Priorities in the range 0-20 are reserved for the C++ compiler. This is to ensure that proper initialization of the C++ run-time system takes place before the execution of your program. The "library" keyword represents a priority of 32 and can be used for class libraries that require initialization before the program is initialized. The "program" keyword represents a priority of 64 and is the default priority for any compiled code. Specifying "before" adjusts the priority by subtracting one. Specifying "after" adjusts the priority by adding one.

A source file containing the following "initialize" pragma specifies that the initialization of static data in the file will take place before initialization of all other static data in the program since a priority of 63 will be assigned.

Example:

```
#pragma initialize before program
```

If we specify "after" instead of "before", the initialization of the static data in the file will occur after initialization of all other static data in the program since a priority of 65 will be assigned.

Note that the following is equivalent to the "before" example

Example:

```
#pragma initialize 63
```

and the following is equivalent to the "after" example.

Example:

```
#pragma initialize 65
```

The use of the "before", "after", and "program" keywords are more descriptive in the intent of the pragmas.

It is recommended that a priority of 32 (the priority used when the "library" keyword is specified) be used when developing class libraries. This will ensure that initialization of static data defined by the class library will take place before initialization of static data defined by the program. The following "initialize" pragma can be used to achieve this.

Example:

```
#pragma initialize library
```

11.16 The INLINE_DEPTH Pragma (C++ Only)

When an in-line function is called, the function call may be replaced by the in-line expansion for that function. This in-line expansion may include calls to other in-line functions which can also be expanded. The "inline_depth" pragma can be used to set the number of times this expansion of in-line functions will occur for a call.

The form of the "inline_depth" pragma is as follows.

```
#pragma inline_depth [( ] n [)] [;]
```

where ***description:***

n is the depth of expansion. If n is 0, no expansion will occur. If n is 1, only the original call is expanded. If n is 2, the original call and the in-line functions invoked by the original function will be expanded. The default value for n is 3. The maximum value for n is 255. Note that no expansion of recursive in-line functions occur unless enabled using the "inline_recursion" pragma.

11.17 The INLINE_RECURSION Pragma (C++ Only)

The "inline_recursion" pragma controls the recursive expansion of inline functions. The form of the "inline_recursion" pragma is as follows.

```
#pragma inline_recursion [(] on | off [)] [;]
```

Specifying "on" will enable expansion of recursive inline functions. The depth of expansion is specified by the "inline_depth" pragma. The default depth is 3. Specifying "off" suppresses expansion of recursive inline functions. This is the default.

11.18 The INTRINSIC Pragma

Certain functions, those listed in the description of the "oi" option, have intrinsic forms. These functions are special functions that are recognized by the compiler and processed in a special way. For example, the compiler may choose to generate in-line code for the function. The intrinsic attribute for these special functions is set by specifying the "oi" option or using an "intrinsic" pragma.

The following describes the form of the "intrinsic" pragma.

```
#pragma intrinsic ( fn {, fn} ) [;]
```

where ***description:***

fn is the name of a function.

Suppose the following source code was compiled without using the "oi" option so that no function had the intrinsic attribute. If we wanted the intrinsic form of the `sin` function to be used, we could specify the function in an "intrinsic" pragma.

```
#include <math.h>
#pragma intrinsic( sin );

double test( double x )
{
    return( sin( x ) );
}
```

11.19 The MESSAGE Pragma

The "message" pragma can be used to issue a message with the specified text to the standard output without terminating compilation. The following describes the form of the "message" pragma.

```
#pragma message ( "message text" ) [;]
```

where description:

"message text" is the text of the message that you wish to display.

The following is an example.

```
#if defined(__386__)
    ...
#else
#pragma message ( "assuming 16-bit compile" );
#endif
```

11.20 The ONCE Pragma

The "once" pragma can be used to indicate that the file which contains this pragma should only be opened and processed "once". The following describes the form of the "once" pragma.

```
#pragma once [;]
```

Assume that the file "foo.h" contains the following text.

Example:

```
#ifndef _FOO_H_INCLUDED
#define _FOO_H_INCLUDED
#pragma once
    .
    .
    .
#endif
```

The first time that the compiler processes "foo.h" and encounters the "once" pragma, it records the file's name. Subsequently, whenever the compiler encounters a `#include` statement that refers to "foo.h", it will not open the include file again. This can help speed up processing of `#include` files and reduce the time required to compile an application.

11.21 The PACK Pragma

The "pack" pragma can be used to control the way in which structures are stored in memory. By default, Watcom C/C++ aligns all structures and its fields on a byte boundary. There are 4 forms of the "pack" pragma.

The following form of the "pack" pragma can be used to change the alignment of structures and their fields in memory.

```
#pragma pack ( n ) [;]
```

where ***description:***

n is 1, 2, 4, 8 or 16 and specifies the method of alignment.

The alignment of structure members is described in the following table. If the size of the member is 1, 2, 4, 8 or 16, the alignment is given for each of the "zp" options. If the member of the structure is an array or structure, the alignment is described by the row "x".

sizeof(member)	zp1	zp2	zp4	zp8	zp16
1	0	0	0	0	0
2	0	2	2	2	2
4	0	2	4	4	4
8	0	2	4	8	8
16	0	2	4	8	16
x	aligned to largest member				

An alignment of 0 means no alignment, 2 means word boundary, 4 means doubleword boundary, etc. If the largest member of structure "x" is 1 byte then "x" is not aligned. If the largest member of structure "x" is 2 bytes then "x" is aligned according to row 2. If the largest member of structure "x" is 4 bytes then "x" is aligned according to row 4. If the largest member of structure "x" is 8 bytes then "x" is aligned according to row 8. If the largest member of structure "x" is 16 bytes then "x" is aligned according to row 16.

If no value is specified in the "pack" pragma, a default value of 8 is used. Note that the default value can be changed with the "zp" Watcom C/C++ compiler command line option.

The following form of the "pack" pragma can be used to save the current alignment amount on an internal stack.

```
#pragma pack ( push ) [;]
```

The following form of the "pack" pragma can be used to save the current alignment amount on an internal stack and set the current alignment.

```
#pragma pack ( push, number ) [;]
```

The following form of the "pack" pragma can be used to restore the previous alignment amount from an internal stack.

```
#pragma pack ( pop ) [;]
```

11.22 The READ_ONLY_FILE Pragma

Explicit listing of dependencies in a makefile can often be tedious in the development and maintenance phases of a project. The Watcom C/C++ compiler will insert dependency information into the object file as it processes source files so that a complete snapshot of the files necessary to build the object file are recorded. The "read_only_file" pragma can be used to prevent the name of the source file that includes it from being included in the dependency information that is written to the object file.

This pragma is commonly used in system header files since they change infrequently (and, when they do, there should be no impact on source files that have included them).

The form of the "read_only_file" pragma follows.

```
#pragma read_only_file [;]
```

For more information on make dependencies, see the section entitled "Automatic Dependency Detection (.AUTODEPEND)" in the ***Watcom C/C++ Tools User's Guide.***

11.23 The TEMPLATE_DEPTH Pragma (C++ Only)

The "template_depth" pragma provides a hard limit for the amount of nested template expansions allowed so that infinite expansion can be detected.

The form of the "template_depth" pragma is as follows.

```
#pragma template_depth [( ] n [)] [;]
```

where ***description:***

n is the depth of expansion. If the value of n is less than 2, if will default to 2. If n is not specified, a warning message will be issued and the default value for n will be 100.

The following example of recursive template expansion illustrates why this pragma can be useful.

Example:

```
#pragma template_depth(10);

template <class T>
struct S {
    S<T*> x;
};

S<char> v;
```

11.24 The WARNING Pragma (C++ Only)

The "warning" pragma sets the level of warning messages. The form of the "warning" pragma is as follows.

```
#pragma warning msg_num level [;]
```

where ***description:***

msg_num is the number of the warning message. This number corresponds to the number issued by the compiler and can be found in the appendix entitled "Watcom C++ Diagnostic Messages" on page 397. If `msg_num` is "*", the level of all warning messages is changed to the specified level. Make sure to strip all leading zeroes from the message number (to avoid interpretation as an octal constant).

level is a number from 0 to 9 and represents the level of the warning message. When a value of zero is specified, the warning becomes an error.

11.25 Auxiliary Pragmas

The following sections describe the capabilities provided by auxiliary pragmas.

11.25.1 Specifying Symbol Attributes

Auxiliary pragmas are used to describe attributes that affect code generation. Initially, the compiler defines a default set of attributes. Each auxiliary pragma refers to one of the following.

1. a symbol (such as a variable or function)
2. a type definition that resolves to a function type
3. the default set of attributes defined by the compiler

When an auxiliary pragma refers to a particular symbol, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to the specified symbol and can only be changed by another auxiliary pragma that refers to the same symbol.

An example of a type definition that resolves to a function type is the following.

```
typedef void (*func_type)();
```

When an auxiliary pragma refers to a such a type definition, a copy of the current set of default attributes is made and merged with the attributes specified in the auxiliary pragma. The resulting attributes are assigned to each function whose type matches the specified type definition.

When "default" is specified instead of a symbol name, the attributes specified by the auxiliary pragma change the default set of attributes. The resulting attributes are used by all symbols that have not been specifically referenced by a previous auxiliary pragma.

Note that all auxiliary pragmas are processed before code generation begins. Consider the following example.

```
code in which symbol x is referenced
#pragma aux y <attrs_1>;
code in which symbol y is referenced
code in which symbol z is referenced
#pragma aux default <attrs_2>;
#pragma aux x <attrs_3>;
```

Auxiliary attributes are assigned to `x, y` and `z` in the following way.

1. Symbol `x` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>` and `<attrs_3>.`

2. Symbol `y` is assigned the initial default attributes merged with the attributes specified by `<attrs_1>.`

3. Symbol `z` is assigned the initial default attributes merged with the attributes specified by `<attrs_2>.`

11.25.2 Alias Names

When a symbol referred to by an auxiliary pragma includes an alias name, the attributes of the alias name are also assumed by the specified symbol.

There are two methods of specifying alias information. In the first method, the symbol assumes only the attributes of the alias name; no additional attributes can be specified. The second method is more general since it is possible to specify an alias name as well as additional auxiliary information. In this case, the symbol assumes the attributes of the alias name as well as the attributes specified by the additional auxiliary information.

The simple form of the auxiliary pragma used to specify an alias is as follows.

```
#pragma aux ( sym, [far16] alias ) [;]
```

where ***description:***

sym is any valid C/C++ identifier.

alias is the alias name and is any valid C/C++ identifier.

The `far16` attribute should only be used on systems that permit the calling of 16-bit code from 32-bit code. Currently, the only supported operating system that allows this is 32-bit OS/2. If you have any libraries of functions or APIs that are only available as 16-bit code and you wish to access these functions and APIs from 32-bit code, you must specify the `far16` attribute. If the `far16` attribute is specified, the compiler will generate special code which allows the 16-bit code to be called from 32-bit code. Note that a `far16` function must be a function whose attributes are those specified by one of the alias names `__cdecl` or `__pascal.` These alias names will be described in a later section.

Consider the following example.

```
#pragma aux push_args parm [] ;
#pragma aux ( rtn, push_args ) ;
```

The routine `rtn` assumes the attributes of the alias name `push_args` which specifies that the arguments to `rtn` are passed on the stack.

Let us look at an example in which the symbol is a type definition.

```
typedef void (func_type)(int);

#pragma aux push_args parm [];
#pragma aux ( func_type, push_args );

extern func_type rtn1;
extern func_type rtn2;
```

The first auxiliary pragma defines an alias name called `push_args` that specifies the mechanism to be used to pass arguments. The mechanism is to pass all arguments on the stack. The second auxiliary pragma associates the attributes specified in the first pragma with the type definition `func_type.` Since `rtn1` and `rtn2` are of type `func_type,` arguments to either of those functions will be passed on the stack.

The general form of an auxiliary pragma that can be used to specify an alias is as follows.

```
#pragma aux ( alias ) sym aux_attrs [;]
```

where ***description:***

alias is the alias name and is any valid C/C++ identifier.

sym is any valid C/C++ identifier.

aux_attrs are attributes that can be specified with the auxiliary pragma.

Consider the following example.

```
#pragma aux HIGH_C "*"                                  \
                   parm caller []                       \
                   value no8087                         \
                   modify [eax ecx edx fs gs];
#pragma aux (HIGH_C) rtn1;
#pragma aux (HIGH_C) rtn2;
#pragma aux (HIGH_C) rtn3;
```

The routines `rtn1, rtn2` and `rtn3` assume the same attributes as the alias name `HIGH_C` which defines the calling convention used by the MetaWare High C compiler. Note that register ES must also be specified in the "modify" register set when using a memory model that is not a small data model. Whenever calls are made to `rtn1, rtn2` and `rtn3,` the MetaWare High C calling convention will be used.

Note that if the attributes of `HIGH_C` change, only one pragma needs to be changed. If we had not used an alias name and specified the attributes in each of the three pragmas for `rtn1, rtn2` and `rtn3,` we would have to change all three pragmas. This approach also reduces the amount of memory required by the compiler to process the source file.

WARNING! The alias name `HIGH_C` is just another symbol. If `HIGH_C` appeared in your source code, it would assume the attributes specified in the pragma for `HIGH_C.`

11.25.3 Predefined Aliases

A number of symbols are predefined by the compiler with a set of attributes that describe a particular calling convention. These symbols can be used as aliases. The following is a list of these symbols.

__cdecl `__cdecl` or `cdecl` defines the calling convention used by Microsoft compilers.

__pascal `__pascal` or `pascal` defines the calling convention used by OS/2 1.x and Windows 3.x API functions.

__stdcall `__stdcall` or `stdcall` defines a special calling convention used by the Win32 API functions.

__syscall `__syscall` or `syscall` defines the calling convention used by the 32-bit OS/2 API functions.

__system `__system` or `system` are identical to `__syscall.`

The following describes the attributes of the above alias names.

11.25.3.1 Predefined "__cdecl" Alias

```
#pragma aux __cdecl "_*" \
          parm caller [] \
          value struct float struct routine [eax] \
          modify [eax ecx edx]
```

Notes:

1. All symbols are preceded by an underscore character.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the called routine allocates space for the return value and returns a pointer to the return value in register EAX.

4. Registers EAX, ECX and EDX are not saved and restored when a call is made.

11.25.3.2 Predefined "__pascal" Alias

```
#pragma aux __pascal "^" \
          parm reverse routine [] \
          value struct float struct caller [] \
          modify [eax ebx ecx edx]
```

Notes:

1. All symbols are mapped to upper case.

2. Arguments are pushed on the stack in reverse order. That is, the first argument is pushed first, the second argument is pushed next, and so on. The routine being called will remove the arguments from the stack.

3. Floating-point values are returned in the same way as structures. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value.

4. Registers EAX, EBX, ECX and EDX are not saved and restored when a call is made.

11.25.3.3 Predefined "__stdcall" Alias

```
#pragma aux __stdcall "_*@nnn" \
          parm routine [] \
          value struct struct caller [] \
          modify [eax ecx edx]
```

Notes:

1. All symbols are preceded by an underscore character.

2. All C symbols (extern "C" symbols in C++) are suffixed by "@nnn" where "nnn" is the sum of the argument sizes (each size is rounded up to a multiple of 4 bytes so that char and short are size 4). When the argument list contains "...", the "@nnn" suffix is omitted.

3. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The called routine will remove the arguments from the stack.

4. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

5. Registers EAX, ECX and EDX are not saved and restored when a call is made.

11.25.3.4 Predefined "__syscall" Alias

```
#pragma aux __syscall "*" \
            parm caller [] \
            value struct struct caller [] \
            modify [eax ecx edx]
```

Notes:

1. Symbols names are not modified, that is, they are not adorned with leading or trailing underscores.

2. Arguments are pushed on the stack from right to left. That is, the last argument is pushed first. The calling routine will remove the arguments from the stack.

3. When a structure is returned, the caller allocates space on the stack. The address of the allocated space will be pushed on the stack immediately before the call instruction. Upon returning from the call, register EAX will contain address of the space allocated for the return value. Floating-point values are returned in 80x87 register ST(0).

4. Registers EAX, ECX and EDX are not saved and restored when a call is made.

11.25.4 Alternate Names for Symbols

The following form of the auxiliary pragma can be used to describe the mapping of a symbol from its source form to its object form.

```
#pragma aux sym obj_name [;]
```

where ***description:***

sym is any valid C/C++ identifier.

obj_name is any character string enclosed in double quotes.

When specifying `obj_name,` the asterisk character ('*') has a special meaning; it is a placeholder for `sym.`

In the following example, the name "myrtn" will be replaced by "myrtn_" in the object file.

```
#pragma aux myrtn "*_";
```

This is the default for all function names.

In the following example, the name "myvar" will be replaced by "_myvar" in the object file.

```
#pragma aux myvar "_*";
```

This is the default for all variable names.

The default mapping for all symbols can also be changed as illustrated by the following example.

```
#pragma aux default "_*_";
```

The above auxiliary pragma specifies that all names will be prefixed and suffixed by an underscore character ('_').

The '^' character also has a special meaning. Whenever it is encountered in `obj_name,` it is replaced by the upper case version of `sym.`

In the following example, the name "myrtn" will be replaced by "MYRTN" in the object file.

```
#pragma aux myrtn "^";
```

11.25.5 Describing Calling Information

The following form of the auxiliary pragma can be used to describe the way a function is to be called.

```
#pragma aux sym far [;]
    or
#pragma aux sym near [;]
    or
#pragma aux sym = in_line [;]

in_line ::= { const | (seg id) | (offset id) | (reloff id)
          | "asm" }
```

where ***description:***

sym is a function name.

const is a valid C/C++ integer constant.

id is any valid C/C++ identifier.

seg specifies the segment of the symbol `id`.

offset specifies the offset of the symbol `id`.

reloff specifies the relative offset of the symbol `id` for near control transfers.

asm is an assembly language instruction or directive.

In the following example, Watcom C/C++ will generate a far call to the function `myrtn`.

```
#pragma aux myrtn far;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a far call will be generated even if you are compiling for a memory model with a small code model.

In the following example, Watcom C/C++ will generate a near call to the function `myrtn`.

```
#pragma aux myrtn near;
```

Note that this overrides the calling sequence that would normally be generated for a particular memory model. In other words, a near call will be generated even if you are compiling for a memory model with a big code model.

In the following DOS example, Watcom C/C++ will generate the sequence of bytes following the "=" character in the auxiliary pragma whenever a call to `mode4` is encountered. `mode4` is called an in-line function.

```
void mode4(void);
#pragma aux mode4 =                 \
    0xb4 0x00       /* mov AH,0 */ \
    0xb0 0x04       /* mov AL,4 */ \
    0xcd 0x10       /* int 10H  */ \
    modify [ AH AL ];
```

The sequence in the above DOS example represents the following lines of assembly language instructions.

```
mov   AH,0       ; select function "set mode"
mov   AL,4       ; specify mode (mode 4)
int   10H        ; BIOS video call
```

The above example demonstrates how to generate BIOS function calls in-line without writing an assembly language function and calling it from your C/C++ program. The C prototype for the function `mode4` is not necessary but is included so that we can take advantage of the argument type checking provided by Watcom C/C++.

The following DOS example is equivalent to the above example but mnemonics for the assembly language instructions are used instead of the binary encoding of the assembly language instructions.

```
void mode4(void);
#pragma aux mode4 =     \
    "mov AH,0",         \
    "mov AL,4",         \
    "int 10H"           \
    modify [ AH AL ];
```

A sequence of in-line assembly language instructions may contain symbolic references. In the following example, a near call to the function `myalias` is made whenever `myrtn` is called.

```
extern void myalias(void);
void myrtn(void);
#pragma aux myrtn =                     \
    0xe8 offset myalias /* near call */;
```

In the following example, a far call to the function `myalias` is made whenever `myrtn` is called.

```
extern void myalias(void);
void myrtn(void);
#pragma aux myrtn =                                    \
    0x9a offset myalias seg myalias /* far call */;
```

11.25.5.1 Loading Data Segment Register

An application may have been compiled so that the segment register DS does not contain the segment address of the default data segment (group "DGROUP"). This is usually the case if you are using a large data memory model. Suppose you wish to call a function that assumes that the segment register DS contains the segment address of the default data segment. It would be very cumbersome if you were forced to compile your application so that the segment register DS contained the default data segment (a small data memory model).

The following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment before calling the specified function.

```
#pragma aux sym parm loadds [;]
```

where ***description:***

sym is a function name.

Alternatively, the following form of the auxiliary pragma will cause the segment register DS to be loaded with the segment address of the default data segment as part of the prologue sequence for the specified function.

```
#pragma aux sym loadds [;]
```

where ***description:***

sym is a function name.

11.25.5.2 Defining Exported Symbols in Dynamic Link Libraries

An exported symbol in a dynamic link library is a symbol that can be referenced by an application that is linked with that dynamic link library. Normally, symbols in dynamic link

libraries are exported using the Watcom Linker "EXPORT" directive. An alternative method is to use the following form of the auxiliary pragma.

```
#pragma aux sym export [;]
```

where ***description:***

sym is a function name.

11.25.5.3 Forcing a Stack Frame

Normally, a function contains a stack frame if arguments are passed on the stack or an automatic variable is allocated on the stack. No stack frame will be generated if the above conditions are not satisfied. The following form of the auxiliary pragma will force a stack frame to be generated under any circumstance.

```
#pragma aux sym frame [;]
```

where ***description:***

sym is a function name.

11.25.6 Describing Argument Information

Using auxiliary pragmas, you can describe the calling convention that Watcom C/C++ is to use for calling functions. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes argument passing is the following.

```
#pragma aux sym parm { pop_info | reverse | {reg_set} } [;]

pop_info ::= caller | routine
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

11.25.6.1 Passing Arguments in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to pass arguments to a particular function.

```
#pragma aux sym parm {reg_set} [;]
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

Register sets establish a priority for register allocation during argument list processing. Register sets are processed from left to right. However, within a register set, registers are chosen in any order. Once all register sets have been processed, any remaining arguments are pushed on the stack.

Note that regardless of the register sets specified, only certain combinations of registers will be selected for arguments of a particular type.

Note that arguments of type **float** and **double** are always pushed on the stack when the "fpi" or "fpi87" option is used.

double Arguments of type **double** can only be passed in one of the following register pairs: EDX:EAX, ECX:EBX, ECX:EAX, ECX:ESI, EDX:EBX, EDI:EAX, ECX:EDI, EDX:ESI, EDI:EBX, ESI:EAX, ECX:EDX, EDX:EDI, EDI:ESI, ESI:EBX or EBX:EAX. For example, if the following register set was specified for a routine having an argument of type **double,**

```
[EBP EBX]
```

the argument would be pushed on the stack since a valid register combination for 8-byte arguments is not contained in the register set. Note that this method for passing arguments of type **double** is supported only when the "fpc" option is used. Note that this argument passing method does not include the passing of 8-byte structures.

far pointer A far pointer can only be passed in one of the following register pairs: DX:EAX, CX:EBX, CX:EAX, CX:ESI, DX:EBX, DI:EAX, CX:EDI, DX:ESI, DI:EBX, SI:EAX, CX:EDX, DX:EDI, DI:ESI, SI:EBX, BX:EAX, FS:ECX, FS:EDX, FS:EDI, FS:ESI, FS:EBX, FS:EAX, GS:ECX, GS:EDX, GS:EDI, GS:ESI, GS:EBX, GS:EAX, DS:ECX, DS:EDX, DS:EDI, DS:ESI, DS:EBX, DS:EAX, ES:ECX, ES:EDX, ES:EDI, ES:ESI, ES:EBX or ES:EAX. For example, if a far pointer is passed to a function with the following register set,

```
[ES EBP]
```

the argument would be pushed on the stack since a valid register combination for a far pointer is not contained in the register set.

int The only registers that will be assigned to 4-byte arguments (e.g., arguments of type **int**) are: EAX, EBX, ECX, EDX, ESI and EDI. For example, if the following register set was specified for a routine with one argument of type **int,**

```
[EBP]
```

the argument would be pushed on the stack since a valid register combination for 4-byte arguments is not contained in the register set. Note that this argument passing method includes 4-byte structures. Note that this argument passing method also includes arguments of type **float** but only when the "fpc" option is used.

char, short int

Arguments whose size is 1 byte or 2 bytes (e.g., arguments of type **char** and **short int** as well as 2-byte structures) are promoted to 4 bytes and are then assigned registers as if they were 4-byte arguments.

others Arguments that do not fall into one of the above categories cannot be passed in registers and are pushed on the stack. Once an argument has been assigned a position on the stack, all remaining arguments will be assigned a position on the stack even if all register sets have not yet been exhausted.

Notes:

1. The default register set is [EAX EBX ECX EDX].

2. Specifying registers AH and AL is equivalent to specifying register AX. Specifying registers DH and DL is equivalent to specifying register DX. Specifying registers CH and CL is equivalent to specifying register CX. Specifying registers BH and BL is equivalent to specifying register BX. Specifying register EAX implies that register AX has been specified. Specifying register EBX implies that register BX has been specified. Specifying register ECX implies that register CX has been specified. Specifying register EDX implies that register DX has been specified. Specifying register EDI implies that register DI has been specified. Specifying register ESI implies that register SI has been specified. Specifying register EBP implies that register BP has been specified. Specifying register ESP implies that register SP has been specified.

3. If you are compiling for a memory model with a small data model, or the "zdp" compiler option is specified, any register combination containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment. Note that the "zdf" compiler option can be used to specify that register DS does not contain that segment address of the program's data segment. In this case, register combinations containing register DS are legal.

4. If you are compiling for the flat memory model, any register combination containing DS or ES becomes illegal. In a flat memory model, code and data reside in the same segment. Segment registers DS and ES point to this segment and must remain unchanged.

Consider the following example.

```
#pragma aux myrtn parm [eax ebx ecx edx] [ebp esi];
```

Suppose `myrtn` is a routine with 3 arguments each of type **double.**

1. The first argument will be passed in the register pair EDX:EAX.
2. The second argument will be passed in the register pair ECX:EBX.
3. The third argument will be pushed on the stack since EBP:ESI is not a valid register pair for arguments of type **double.**

It is possible for registers from the second register set to be used before registers from the first register set are used. Consider the following example.

```
#pragma aux myrtn parm [eax ebx ecx edx] [esi edi];
```

Suppose `myrtn` is a routine with 3 arguments, the first of type **int** and the second and third of type **double.**

1. The first argument will be passed in the register EAX.
2. The second argument will be passed in the register pair ECX:EBX.
3. The third argument will be passed in the register set EDI:ESI.

Note that registers are no longer selected from a register set after registers are selected from subsequent register sets, even if all registers from the original register set have not been exhausted.

An empty register set is permitted. All subsequent register sets appearing after an empty register set are ignored; all remaining arguments are pushed on the stack.

Notes:

1. If a single empty register set is specified, all arguments are passed on the stack.

2. If no register set is specified, the default register set [EAX EBX ECX EDX] is used.

11.25.6.2 Forcing Arguments into Specific Registers

It is possible to force arguments into specific registers. Suppose you have a function, say "mycopy", that copies data. The first argument is the source, the second argument is the destination, and the third argument is the length to copy. If we want the first argument to be passed in the register ESI, the second argument to be passed in register EDI and the third argument to be passed in register ECX, the following auxiliary pragma can be used.

```
void mycopy( char near *, char *, int );
#pragma aux mycopy parm [EDI] [ESI] [ECX];
```

Note that you must be aware of the size of the arguments to ensure that the arguments get passed in the appropriate registers.

11.25.6.3 Passing Arguments to In-Line Functions

For functions whose code is generated by Watcom C/C++ and whose argument list is described by an auxiliary pragma, Watcom C/C++ has some freedom in choosing how arguments are assigned to registers. Since the code for in-line functions is specified by the programmer, the description of the argument list must be very explicit. To achieve this, Watcom C/C++ assumes that each register set corresponds to an argument. Consider the following DOS example of an in-line function called `scrollactivepgup.`

```
void scrollactivepgup(char,char,char,char,char,char);
#pragma aux scrollactivepgup = \
    "mov AH,6"   \
    "int 10h"    \
    parm [ch] [cl] [dh] [dl] [al] [bh] \
    modify [ah];
```

The BIOS video call to scroll the active page up requires the following arguments.

1. The row and column of the upper left corner of the scroll window is passed in registers CH and CL respectively.

2. The row and column of the lower right corner of the scroll window is passed in registers DH and DL respectively.

3. The number of lines blanked at the bottom of the window is passed in register AL.

4. The attribute to be used on the blank lines is passed in register BH.

When passing arguments, Watcom C/C++ will convert the argument so that it fits in the register(s) specified in the register set for that argument. For example, in the above example, if the first argument to `scrollactivepgup` was called with an argument whose type was **int**, it would first be converted to **char** before assigning it to register CH. Similarly, if an in-line function required its argument in register EAX and the argument was of type **short int**, the argument would be converted to **long int** before assigning it to register EAX.

In general, Watcom C/C++ assigns the following types to register sets.

1. A register set consisting of a single 8-bit register (1 byte) is assigned a type of **unsigned char.**

2. A register set consisting of a single 16-bit register (2 bytes) is assigned a type of **unsigned short int.**

3. A register set consisting of a single 32-bit register (4 bytes) is assigned a type of **unsigned long int.**

4. A register set consisting of two 32-bit registers (8 bytes) is assigned a type of **double.**

11.25.6.4 Removing Arguments from the Stack

The following form of the auxiliary pragma specifies who removes from the stack arguments that were pushed on the stack.

```
#pragma aux sym parm (caller | routine) [;]
```

where ***description:***

sym is a function name.

"caller" specifies that the caller will pop the arguments from the stack; "routine" specifies that the called routine will pop the arguments from the stack. If "caller" or "routine" is omitted, "routine" is assumed unless the default has been changed in a previous auxiliary pragma, in which case the new default is assumed.

11.25.6.5 Passing Arguments in Reverse Order

The following form of the auxiliary pragma specifies that arguments are passed in the reverse order.

```
#pragma aux sym parm reverse [;]
```

where ***description:***

sym is a function name.

Normally, arguments are processed from left to right. The leftmost arguments are passed in registers and the rightmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from right to left.

When arguments are reversed, the rightmost arguments are passed in registers and the leftmost arguments are passed on the stack (if the registers used for argument passing have been exhausted). Arguments that are passed on the stack are pushed from left to right.

Reversing arguments is most useful for functions that require arguments to be passed on the stack in an order opposite from the default. The following auxiliary pragma demonstrates such a function.

```
#pragma aux rtn parm reverse [];
```

11.25.7 Describing Function Return Information

Using auxiliary pragmas, you can describe the way functions are to return values. This is particularly useful when interfacing to functions that have been compiled by other compilers or functions written in other programming languages.

The general form of an auxiliary pragma that describes the way a function returns its value is the following.

```
#pragma aux sym value {no8087 | reg_set | struct_info} [;]
struct_info ::= struct {float | struct | (routine | caller) | reg_set}
```

where ***description:***

sym is a function name.

reg_set is called a register set. The register sets specify the registers that are to be used for argument passing. A register set is a list of registers separated by spaces and enclosed in square brackets.

11.25.7.1 Returning Function Values in Registers

The following form of the auxiliary pragma can be used to specify the registers that are to be used to return a function's value.

```
#pragma aux sym value reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

Note that the method described below for returning values of type **float** or **double** is supported only when the "fpc" option is used.

Depending on the type of the return value, only certain registers are allowed in *reg_set*.

1-byte For 1-byte return values, only the following registers are allowed: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2-byte For 2-byte return values, only the following registers are allowed: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

4-byte For 4-byte return values (including near pointers), only the following register are allowed: EAX, EDX, EBX, ECX, ESI or EDI. If no register set is specified, register EAX will be used. This form of the auxiliary pragma is legal for functions of type **float** when using the "fpc" option only.

far pointer For functions that return far pointers, the following register pairs are allowed: DX:EAX, CX:EBX, CX:EAX, CX:ESI, DX:EBX, DI:EAX, CX:EDI, DX:ESI, DI:EBX, SI:EAX, CX:EDX, DX:EDI, DI:ESI, SI:EBX, BX:EAX, FS:ECX, FS:EDX, FS:EDI, FS:ESI, FS:EBX, FS:EAX, GS:ECX, GS:EDX, GS:EDI, GS:ESI, GS:EBX, GS:EAX, DS:ECX, DS:EDX, DS:EDI, DS:ESI, DS:EBX, DS:EAX, ES:ECX, ES:EDX, ES:EDI, ES:ESI, ES:EBX or ES:EAX. If no register set is specified, the registers DX:EAX will be used.

8-byte For 8-byte return values (including functions of type **double**), only the following register pairs are allowed: EDX:EAX, ECX:EBX, ECX:EAX, ECX:ESI, EDX:EBX, EDI:EAX, ECX:EDI, EDX:ESI, EDI:EBX, ESI:EAX, ECX:EDX, EDX:EDI, EDI:ESI, ESI:EBX or EBX:EAX. If no register set is specified, the registers EDX:EAX will be used. This form of the auxiliary pragma is legal for functions of type **double** when using the "fpc" option only.

Notes:

1. An empty register set is not allowed.

2. If you are compiling for a memory model which has a small data model, any of the above register combinations containing register DS becomes illegal. In a small data model, segment register DS must remain unchanged as it points to the program's data segment.

3. If you are compiling for the flat memory model, any register combination containing DS or ES becomes illegal. In a flat memory model, code and data reside in the same segment. Segment registers DS and ES point to this segment and must remain unchanged.

11.25.7.2 Returning Structures

Typically, structures are not returned in registers. Instead, the caller allocates space on the stack for the return value and sets register ESI to point to it. The called routine then places the return value at the location pointed to by register ESI.

The following form of the auxiliary pragma can be used to specify the register that is to be used to point to the return value.

```
#pragma aux sym value struct (caller|routine) reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

"caller" specifies that the caller will allocate memory for the return value. The address of the memory allocated for the return value is placed in the register specified in the register set by the caller before the function is called. If an empty register set is specified, the address of the memory allocated for the return value will be pushed on the stack immediately before the call and will be returned in register EAX by the called routine.

"routine" specifies that the called routine will allocate memory for the return value. Upon returning to the caller, the register specified in the register set will contain the address of the return value. An empty register set is not allowed.

Only the following registers are allowed in the register set: EAX, EDX, EBX, ECX, ESI or EDI. Note that in a big data model, the address in the return register is assumed to be in the segment specified by the value in the SS segment register.

If the size of the structure being returned is 1, 2 or 4 bytes, it will be returned in registers. The return register will be selected from the register set in the following way.

1. A 1-byte structure will be returned in one of the following registers: AL, AH, DL, DH, BL, BH, CL or CH. If no register set is specified, register AL will be used.

2. A 2-byte structure will be returned in one of the following registers: AX, DX, BX, CX, SI or DI. If no register set is specified, register AX will be used.

3. A 4-byte structure will be returned in one of the following registers: EAX, EDX, EBX, ECX, ESI or EDI. If no register set is specified, register EAX will be used.

The following form of the auxiliary pragma can be used to specify that structures whose size is 1, 2 or 4 bytes are not to be returned in registers. Instead, the caller will allocate space on the stack for the structure return value and point register ESI to it.

```
#pragma aux sym value struct struct [;]
```

where ***description:***

sym is a function name.

11.25.7.3 Returning Floating-Point Data

There are a few ways available for specifying how the value for a function whose type is **float** or **double** is to be returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in registers. Instead, the caller will allocate space on the stack for the return value and point register ESI to it.

```
#pragma aux sym value struct float [;]
```

where ***description:***

sym is a function name.

In other words, floating-point values are to be returned in the same way structures are returned.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are not to be returned in 80x87 registers when compiling with the "fpi" or "fpi87" option. Instead, the value will be returned in 80x86 registers. This is the default behaviour for the "fpc" option. Function return values whose type is **float** will be returned in register EAX. Function return values whose type is **double** will be returned in registers EDX:EAX. This is the default method for the "fpc" option.

```
#pragma aux sym value no8087 [;]
```

where ***description:***

sym is a function name.

The following form of the auxiliary pragma can be used to specify that function return values whose type is **float** or **double** are to be returned in ST(0) when compiling with the "fpi" or "fpi87" option. This form of the auxiliary pragma is not legal for the "fpc" option.

```
#pragma aux sym value [8087] [;]
```

where ***description:***

sym is a function name.

11.25.8 A Function that Never Returns

The following form of the auxiliary pragma can be used to describe a function that does not return to the caller.

```
#pragma aux sym aborts [;]
```

where ***description:***

sym is a function name.

Consider the following example.

```
#pragma aux exitrtn aborts;
extern void exitrtn(void);

void rtn()
  {
    exitrtn();
  }
```

`exitrtn` is defined to be a function that does not return. For example, it may call `exit` to return to the system. In this case, Watcom C/C++ generates a "jmp" instruction instead of a "call" instruction to invoke `exitrtn.`

11.25.9 Describing How Functions Use Memory

The following form of the auxiliary pragma can be used to describe a function that does not modify any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux sym modify nomemory [;]
```

where ***description:***

sym is a function name.

Consider the following example.

```
#pragma off (check_stack);

extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
    while( i < 10000 ) {
        i += 383;
    }
    myrtn();
    i += 13143;
};
```

To compile the above program, "rtn.c", we issue the following command.

```
C>wcc rtn /oai /d1
C>wpp rtn /oai /d1
C>wcc386 rtn /oai /d1
C>wpp386 rtn /oai /d1
```

For illustrative purposes, we omit loop optimizations from the list of code optimizations that we want the compiler to perform. The "d1" compiler option is specified so that the object file produced by Watcom C/C++ contains source line information.

We can generate a file containing a disassembly of `RTN.OBJ` by issuing the following command.

```
C>wdis rtn /l /s /r
```

The "s" option is specified so that the listing file produced by the Watcom Disassembler contains source lines taken from `RTN.C`. The listing file `RTN.LST` appears as follows.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  00000036 bytes

#pragma off (check_stack);

extern void myrtn(void);

int i = { 1033 };
```

```
extern Rtn() {
 0000  52                Rtn_            push    EDX
 0001  8b 15 00 00 00 00                  mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1               cmp     EDX,00002710H
 000d  7d 08                              jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                  add     EDX,0000017fH
 0015  eb f0                              jmp     L1

    myrtn();
 0017  89 15 00 00 00 00 L2               mov     _i,EDX
 001d  e8 00 00 00 00                     call    myrtn_
 0022  8b 15 00 00 00 00                  mov     EDX,_i

    i += 13143;
 0028  81 c2 57 33 00 00                  add     EDX,00003357H
 002e  89 15 00 00 00 00                  mov     _i,EDX

}
 0034  5a                                 pop     EDX
 0035  c3                                 ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00             _i              - ....

No disassembly errors

------------------------------------------------------------
```

Let us add the following auxiliary pragma to the source file.

```
#pragma aux myrtn modify nomemory;
```

If we compile the source file with the above pragma and disassemble the object file using the Watcom Disassembler, we get the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  00000030 bytes

#pragma off (check_stack);
#pragma aux myrtn modify nomemory;
```

```
extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
 0000  52                Rtn_            push    EDX
 0001  8b 15 00 00 00 00                  mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1               cmp     EDX,00002710H
 000d  7d 08                              jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                  add     EDX,0000017fH
 0015  eb f0                              jmp     L1

    myrtn();
 0017  89 15 00 00 00 00 L2               mov     _i,EDX
 001d  e8 00 00 00 00                     call    myrtn_

    i += 13143;
 0022  81 c2 57 33 00 00                  add     EDX,00003357H
 0028  89 15 00 00 00 00                  mov     _i,EDX

}
 002e  5a                                 pop     EDX
 002f  c3                                 ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00             _i              - ....

No disassembly errors

------------------------------------------------------------
```

Notice that the value of `i` is in register EDX after completion of the "while" loop. After the call to `myrtn,` the value of `i` is not loaded from memory into a register to perform the final addition. The auxiliary pragma informs the compiler that `myrtn` does not modify any memory (i.e., global or static variables) that is used directly or indirectly by `Rtn` and hence register EDX contains the correct value of `i.`

The preceding auxiliary pragma deals with routines that modify memory. Let us consider the case where routines reference memory. The following form of the auxiliary pragma can be used to describe a function that does not reference any memory (i.e., global or static variables) that is used directly or indirectly by the caller.

```
#pragma aux sym parm nomemory modify nomemory [;]
```

where ***description:***

sym is a function name.

Notes:

1. You must specify both "parm nomemory" and "modify nomemory".

Let us replace the auxiliary pragma in the above example with the following auxiliary pragma.

```
#pragma aux myrtn parm nomemory modify nomemory;
```

If you now compile our source file and disassemble the object file using WDIS, the result is the following listing file.

```
Module: rtn.c
Group: 'DGROUP' CONST,_DATA

Segment: '_TEXT' BYTE USE32  0000002a bytes

#pragma off (check_stack);
#pragma aux myrtn parm nomemory modify nomemory;

extern void myrtn(void);

int i = { 1033 };

extern Rtn() {
 0000  52                Rtn_            push    EDX
 0001  8b 15 00 00 00 00                 mov     EDX,_i

    while( i < 10000 ) {
 0007  81 fa 10 27 00 00 L1              cmp     EDX,00002710H
 000d  7d 08                             jge     L2

        i += 383;
    }
 000f  81 c2 7f 01 00 00                 add     EDX,0000017fH
 0015  eb f0                             jmp     L1
```

```
    myrtn();
 0017  e8 00 00 00 00    L2              call    myrtn_

    i += 13143;
 001c  81 c2 57 33 00 00                 add     EDX,00003357H
 0022  89 15 00 00 00 00                 mov     _i,EDX

}
 0028  5a                                pop     EDX
 0029  c3                                ret

No disassembly errors

------------------------------------------------------------

Segment: '_DATA' WORD USE32  00000004 bytes
 0000  09 04 00 00             _i              - ....

No disassembly errors

------------------------------------------------------------
```

Notice that after completion of the "while" loop we did not have to update `i` with the value in register EDX before calling `myrtn.` The auxiliary pragma informs the compiler that `myrtn` does not reference any memory (i.e., global or static variables) that is used directly or indirectly by `myrtn` so updating `i` was not necessary before calling `myrtn.`

11.25.10 Describing the Registers Modified by a Function

The following form of the auxiliary pragma can be used to describe the registers that a function will use without saving.

```
#pragma aux sym modify [exact] reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

Specifying a register set informs Watcom C/C++ that the registers belonging to the register set are modified by the function. That is, the value in a register before calling the function is different from its value after execution of the function.

Registers that are used to pass arguments are assumed to be modified and hence do not have to be saved and restored by the called function. Also, since the EAX register is frequently used to return a value, it is always assumed to be modified. If necessary, the caller will contain code to save and restore the contents of registers used to pass arguments. Note that saving and restoring the contents of these registers may not be necessary if the called function does not modify them. The following form of the auxiliary pragma can be used to describe exactly those registers that will be modified by the called function.

```
#pragma aux sym modify exact reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set.

The above form of the auxiliary pragma tells Watcom C/C++ not to assume that the registers used to pass arguments will be modified by the called function. Instead, only the registers specified in the register set will be modified. This will prevent generation of the code which unnecessarily saves and restores the contents of the registers used to pass arguments.

Also, any registers that are specified in the `value` register set are assumed to be unmodified unless explicitly listed in the `exact` register set. In the following example, the code generator will not generate code to save and restore the value of the stack pointer register since we have told it that "GetSP" does not modify any register whatsoever.

Example:

```
unsigned GetSP(void);
#if defined(__386__)
#pragma aux GetSP = value [esp] modify exact [];
#else
#pragma aux GetSP = value [sp] modify exact [];
#endif
```

11.25.11 An Example

As mentioned in an earlier section, the following pragma defines the calling convention for functions compiled by MetaWare's High C compiler.

```
#pragma aux HIGH_C "*"                                  \
                   parm caller []                       \
                   value no8087                         \
                   modify [eax ecx edx fs gs];
```

Note that register ES must also be specified in the "modify" register set when using a memory model with a non-small data model. Let us discuss this pragma in detail.

""* specifies that all function and variable names appear in object form as they do in source form.

parm caller [] specifies that all arguments are to be passed on the stack (an empty register set was specified) and the caller will remove the arguments from the stack.

value no8087 specifies that floating-point values are to be returned using 80x86 registers and not 80x87 floating-point registers.

modify [eax ecx edx fs gs] specifies that registers EAX, ECX, EDX, FS and GS are not preserved by the called routine.

Note that the default method of returning integer values is used; 1-byte characters are returned in register AL, 2-byte integers are returned in register AX, and 4-byte integers are returned in register EAX.

11.25.12 Auxiliary Pragmas and the 80x87

This section deals with those aspects of auxiliary pragmas that are specific to the 80x87. The discussion in this chapter assumes that one of the "fpi" or "fpi87" options is used to compile functions. The following areas are affected by the use of these options.

1. passing floating-point arguments to functions,
2. returning floating-point values from functions and
3. which 80x87 floating-point registers are allowed to be modified by the called routine.

11.25.12.1 Using the 80x87 to Pass Arguments

By default, floating-point arguments are passed on the 80x86 stack. The 80x86 registers are never used to pass floating-point arguments when a function is compiled with the "fpi" or "fpi87" option. However, they can be used to pass arguments whose type is not floating-point such as arguments of type "int".

The following form of the auxiliary pragma can be used to describe the registers that are to be used to pass arguments to functions.

```
#pragma aux sym parm {reg_set} [;]
```

where ***description:***

sym is a function name.

reg_set is a register set. The register set can contain 80x86 registers and/or the string "8087".

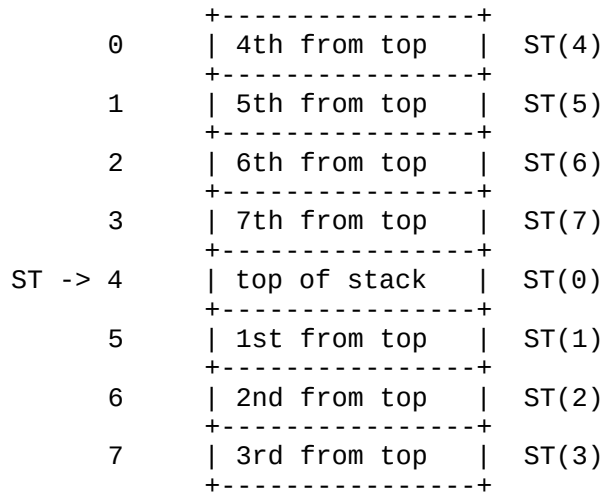
Notes:

1. If an empty register set is specified, all arguments, including floating-point arguments, will be passed on the 80x86 stack.

When the string "8087" appears in a register set, it simply means that floating-point arguments can be passed in 80x87 floating-point registers if the source file is compiled with the "fpi" or "fpi87" option. Before discussing argument passing in detail, some general notes on the use of the 80x87 floating-point registers are given.

The 80x87 contains 8 floating-point registers which essentially form a stack. The stack pointer is called ST and is a number between 0 and 7 identifying which 80x87 floating-point register is at the top of the stack. ST is initially 0. 80x87 instructions reference these registers by specifying a floating-point register number. This number is then added to the current value of ST. The sum (taken modulo 8) specifies the 80x87 floating-point register to be used. The notation ST(n), where "n" is between 0 and 7, is used to refer to the position of an 80x87 floating-point register relative to ST.

When a floating-point value is loaded onto the 80x87 floating-point register stack, ST is decremented (modulo 8), and the value is loaded into ST(0). When a floating-point value is stored and popped from the 80x87 floating-point register stack, ST is incremented (modulo 8) and ST(1) becomes ST(0). The following illustrates the use of the 80x87 floating-point registers as a stack, assuming that the value of ST is 4 (4 values have been loaded onto the 80x87 floating-point register stack).



Starting with version 9.5, the Watcom compilers use all eight of the 80x87 registers as a stack. The initial state of the 80x87 register stack is empty before a program begins execution.

Note: For compatibility with code compiled with version 9.0 and earlier, you can compile with the "fpr" option. In this case only four of the eight 80x87 registers are used as a stack. These four registers were used to pass arguments. The other four registers form what was called the 80x87 cache. The cache was used for local floating-point variables. The state of the 80x87 registers before a program began execution was as follows.

1. The four 80x87 floating-point registers that form the stack are uninitialized.
2. The four 80x87 floating-point registers that form the 80x87 cache are initialized with zero.

Hence, initially the 80x87 cache was comprised of ST(0), ST(1), ST(2) and ST(3). ST had the value 4 as in the above diagram. When a floating-point value was pushed on the stack (as is the case when passing floating-point arguments), it became ST(0) and the 80x87 cache was comprised of ST(1), ST(2), ST(3) and ST(4). When the 80x87 stack was full, ST(0), ST(1), ST(2) and ST(3) formed the stack and ST(4), ST(5), ST(6) and ST(7) formed the 80x87 cache. Version 9.5 and later no longer use this strategy.

The rules for passing arguments are as follows.

1. If the argument is not floating-point, use the procedure described earlier in this chapter.

2. If the argument is floating-point, and a previous argument has been assigned a position on the 80x86 stack (instead of the 80x87 stack), the floating-point argument is also assigned a position on the 80x86 stack. Otherwise proceed to the next step.

3. If the string "8087" appears in a register set in the pragma, and if the 80x87 stack is not full, the floating-point argument is assigned floating-point register ST(0) (the top element of the 80x87 stack). The previous top element (if there was one) is now in ST(1). Since arguments are pushed on the stack from right to left, the leftmost floating-point argument will be in ST(0). Otherwise the floating-point argument is assigned a position on the 80x86 stack.

Consider the following example.

```
#pragma aux myrtn parm [8087];

void main()
{
    float    x;
    double   y;
    int      i;
    long int j;

    x = 7.7;
    i = 7;
    y = 77.77;
    j = 77;
    myrtn( x, i, y, j );
}
```

`myrtn` is an assembly language function that requires four arguments. The first argument of type **float** (4 bytes), the second argument is of type **int** (4 bytes), the third argument is of type **double** (8 bytes) and the fourth argument is of type **long int** (4 bytes). These arguments will be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float,** will be passed in an 80x87 floating-point register.

2. The second argument will be passed on the stack since no 80x86 registers were specified in the register set.

3. The third argument will also be passed on the stack. Remember the following rule: once an argument is assigned a position on the stack, all remaining arguments will be assigned a position on the stack. Note that the above rule holds even though there are some 80x87 floating-point registers available for passing floating-point arguments.

4. The fourth argument will also be passed on the stack.

Let us change the auxiliary pragma in the above example as follows.

```
#pragma aux myrtn parm [eax 8087];
```

The arguments will now be passed to `myrtn` in the following way.

1. Since "8087" was specified in the register set, the first argument, being of type **float** will be passed in an 80x87 floating-point register.

2. The second argument will be passed in register EAX, exhausting the set of available 80x86 registers for argument passing.

3. The third argument, being of type **double**, will also be passed in an 80x87 floating-point register.

4. The fourth argument will be passed on the stack since no 80x86 registers remain in the register set.

11.25.12.2 Using the 80x87 to Return Function Values

The following form of the auxiliary pragma can be used to describe a function that returns a floating-point value in ST(0).

```
#pragma aux sym value reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set containing the string "8087", i.e. [8087].

11.25.12.3 Preserving 80x87 Floating-Point Registers Across Calls

The code generator assumes that all eight 80x87 floating-point registers are available for use within a function unless the "fpr" option is used to generate backward compatible code (older Watcom compilers used four registers as a cache). The following form of the auxiliary pragma specifies that the floating-point registers in the 80x87 cache may be modified by the specified function.

```
#pragma aux sym modify reg_set [;]
```

where ***description:***

sym is a function name.

reg_set is a register set containing the string "8087", i.e. [8087].

This instructs Watcom C/C++ to save any local variables that are located in the 80x87 cache before calling the specified routine.

In-line Assembly Language

12 In-line Assembly Language

The chapters entitled "16-bit Pragmas" on page 161 and "32-bit Pragmas" on page 247 briefly describe the use of the auxiliary pragma to create a sequence of assembly language instructions that can be placed anywhere executable C/C++ statements can appear in your source code. This chapter is devoted to an in-depth look at in-line assembly language programming.

The reasons for resorting to in-line assembly code are varied:

- Speed - You may be interested in optimizing a heavily-used section of code.
- Size - You may wish to optimize a module for size by replacing a library function call with a direct system call.
- Architecture - You may want to access certain features of the Intel x86 architecture that cannot be done so with C/C++ statements.

There are also some reasons for not resorting to in-line assembly code.

- Portability - The code is not portable to different architectures.
- Optimization - Sometimes an optimizing compiler can do a better job of arranging the instruction stream so that it is optimal for a particular processor (such as the 486 or Pentium).

12.1 In-line Assembly Language Tutorial

Doing in-line assembly is reasonably straight-forward with Watcom C/C++ although care must be exercised. You can generate a sequence of in-line assembly anywhere in your C/C++ code stream. The first step is to define the sequence of instructions that you wish to place in-line. The auxiliary pragma is used to do this. Here is a simple example based on a DOS function call that returns a far pointer to the Double-Byte Character Set (DBCS) encoding table.

Example:

```
extern unsigned short far *dbcs_table( void );
#pragma aux dbcs_table = \
        "mov ax,6300h"   \
        "int 21h"        \
        value   [ds si]  \
        modify  [ax];
```

To set up the DOS call, the AH register must contain the hexadecimal value "63" (63h). A DOS function call is invoked by interrupt 21h. DOS returns a far pointer in DS:SI to a table of byte pairs in the form (start of range, end of range). On a non-DBCS system, the first pair will be (0,0). On a Japanese DBCS system, the first pair will be (81h,9Fh).

With each pragma, we define a corresponding function prototype that explains the behaviour of the function in terms of C/C++. Essentially, it is a function that does not take any arguments and that returns a far pointer to a unsigned short item.

The pragma indicates that the result of this "function" is returned in DS:SI (value [ds si]). The pragma also indicates that the AX register is modified by the sequence of in-line assembly code (modify [ax]).

Having defined our in-line assembly code, let us see how it is used in actual C code.

Example:

```
#include <stdio.h>

extern unsigned short far *dbcs_table( void );
#pragma aux dbcs_table = \
        "mov ax,6300h"   \
        "int 21h"        \
        value   [ds si]  \
        modify  [ax];

void main()
{
    if( *dbcs_table() != 0 ) {
        /*
            we are running on a DOS system that
            supports double-byte characters
        */
        printf( "DBCS supported\n" );
    }
}
```

Before you attempt to compile and run this example, consider this: The program will not work! At least, it will not work in most 16-bit memory models. And it doesn't work at all in 32-bit protected mode using a DOS extender. What is wrong with it?

We can examine the disassembled code for this program in order to see why it does not always work in 16-bit real-mode applications.

```
        if( *dbcs_table() != 0 ) {
            /*
                we are running on a DOS system that
                supports double-byte characters
            */
0007  b8 00 63                          mov     ax,6300H
000a  cd 21                             int     21H
000c  83 3c 00                          cmp     word ptr [si],0000H
000f  74 0a                             je      L1

            printf( "DBCS supported\n" );
        }
0011  be 00 00                          mov     si,offset L2
0014  56                                push    si
0015  e8 00 00                          call    printf_
0018  83 c4 02                          add     sp,0002H

    }
```

After the DOS interrupt call, the DS register has been altered and the code generator does nothing to recover the previous value. In the small memory model, the contents of the DS register never change (and any code that causes a change to DS must save and restore its value). It is the programmer's responsibility to be aware of the restrictions imposed by certain memory models especially with regards to the use of segmentation registers. So we must make a small change to the pragma.

```
extern unsigned short far *dbcs_table( void );
#pragma aux dbcs_table = \
        "push ds"        \
        "mov ax,6300h"   \
        "int 21h"        \
        "mov di,ds"      \
        "pop ds"         \
        value   [di si]  \
        modify  [ax];
```

If we compile and run this example with a 16-bit compiler, it will work properly. We can examine the disassembled code for this revised program.

```
    if( *dbcs_table() != 0 ) {
        /*
            we are running on a DOS system that
            supports double-byte characters
        */
0008  1e                              push    ds
0009  b8 00 63                        mov     ax,6300H
000c  cd 21                           int     21H
000e  8c df                           mov     di,ds
0010  1f                              pop     ds
0011  8e c7                           mov     es,di
0013  26 83 3c 00                     cmp     word ptr es:[si],0000H
0017  74 0a                           je      L1

        printf( "DBCS supported\n" );
    }
0019  be 00 00                        mov     si,offset L2
001c  56                              push    si
001d  e8 00 00                        call    printf_
0020  83 c4 02                        add     sp,0002H
```

If you examine this code, you can see that the DS register is saved and restored by the in-line assembly code. The code generator, having been informed that the far pointer is returned in (DI:SI), loads up the ES register from DI in order to reference the far data correctly.

That takes care of the 16-bit real-mode case. What about 32-bit protected mode? When using a DOS extender, you must examine the accompanying documentation to see if the system call that you wish to make is supported by the DOS extender. One of the reasons that this particular DOS call is not so clear-cut is that it returns a 16-bit real-mode segment:offset pointer. A real-mode pointer must be converted by the DOS extender into a protected-mode pointer in order to make it useful. As it turns out, neither the Tenberry Software DOS/4G(W) nor Phar Lap DOS extenders support this particular DOS call (although others may). The issues with each DOS extender are complex enough that the relative merits of using in-line assembly code are not worth it. We present an excerpt from the final solution to this problem.

Example:

```
#ifndef __386__

extern unsigned short far *dbcs_table( void );
#pragma aux dbcs_table = \
        "push ds"        \
        "mov ax,6300h"   \
        "int 21h"        \
        "mov di,ds"      \
        "pop ds"         \
        value   [di si]  \
        modify  [ax];
```

```
#else

unsigned short far * dbcs_table( void )
{
    union REGPACK       regs;
    static short        dbcs_dummy = 0;

    memset( &regs, 0, sizeof( regs ) );
    if( _IsPharLap() ) {
        PHARLAP_block pblock;

        memset( &pblock, 0, sizeof( pblock ) );
        pblock.real_eax = 0x6300;       /* get DBCS vector table */
        pblock.int_num = 0x21;          /* DOS call */
        regs.x.eax = 0x2511;            /* issue real-mode interrupt */
        regs.x.edx = FP_OFF( &pblock ); /* DS:EDX -> parameter block */
        regs.w.ds = FP_SEG( &pblock );
        intr( 0x21, &regs );
        return( firstmeg( pblock.real_ds, regs.w.si ) );
    } else if( _IsDOS4G() ) {
        DPMI_block dblock;

        memset( &dblock, 0, sizeof( dblock ) );
        dblock.eax = 0x6300;            /* get DBCS vector table */
        regs.w.ax = 0x300;              /* DPMI Simulate R-M intr */
        regs.h.bl = 0x21;               /* DOS call */
        regs.h.bh = 0;                  /* flags */
        regs.w.cx = 0;                  /* # bytes from stack */
        regs.x.edi = FP_OFF( &dblock );
        regs.x.es = FP_SEG( &dblock );
        intr( 0x31, &regs );
        return( firstmeg( dblock.ds, dblock.esi ) );
    } else {
        return( &dbcs_dummy );
    }
}

#endif
```

The 16-bit version will use in-line assembly code but the 32-bit version will use a C function that has been crafted to work with both Tenberry Software DOS/4G(W) and Phar Lap DOS extenders. The `firstmeg` function used in the example is shown below.

```
#define REAL_SEGMENT    0x34

void far *firstmeg( unsigned segment, unsigned offset )
{
    void far    *meg1;

    if( _IsDOS4G() ) {
        meg1 = MK_FP( FP_SEG( &meg1 ), ( segment << 4 ) + offset );
    } else {
        meg1 = MK_FP( REAL_SEGMENT, ( segment << 4 ) + offset );
    }
    return( meg1 );
}
```

We have taken a brief look at two features of the auxiliary pragma, the "modify" and "value" attributes.

The "modify" attribute describes those registers that are modified by the execution of the sequence of in-line code. You usually have two choices here; you can save/restore registers that are affected by the code sequence in which case they need not appear in the modify list or you can let the code generator handle the fact that the registers are modified by the code sequence. When you invoke a system function (such as a DOS or BIOS call), you should be careful about any side effects that the call has on registers. If a register is modified by a call and you have not listed it in the modify list or saved/restored it, this can have a disastrous affect on the rest of the code in the function where you are including the in-line code.

The "value" attribute describes the register or registers in which a value is returned (we use the term "returned", not in the sense that a function returns a value, but in the sense that a result is available after execution of the code sequence).

This leads the discussion into the third feature of the auxiliary pragma, the feature that allows us to place the results of C expressions into specific registers as part of the "setup" for the sequence of in-line code. To illustrate this, let us look at another example.

Example:

```
extern void BIOSSetCurPos( unsigned short __rowcol,
                           unsigned char __page );
#pragma aux BIOSSetCurPos =     \
        "push bp"               \
        "mov ah,2"              \
        "int 10h"               \
        "pop bp"                \
        parm   [dx] [bh]        \
        modify [ah];
```

The "parm" attribute specifies the list of registers into which values are to be placed as part of the prologue to the in-line code sequence. In the above example, the "set cursor position" function requires three pieces of information. It requires that the cursor row value be placed in the DH register, that the cursor column value be placed in the DL register, and that the screen page number be placed in the BH register. In this example, we have decided to combine the row and column information into a single "argument" to the function. Note that the function prototype for `BIOSSetCurPos` is important. It describes the types and number of arguments to be set up for the in-line code. It also describes the type of the return value (in this case there is none).

Once again, having defined our in-line assembly code, let us see how it is used in actual C code.

Example:

```
#include <stdio.h>

extern void BIOSSetCurPos( unsigned short __rowcol,
                           unsigned char __page );
#pragma aux BIOSSetCurPos =     \
        "push bp"               \
        "mov ah,2"              \
        "int 10h"               \
        "pop bp"                \
        parm   [dx] [bh]        \
        modify [ah];

void main()
{
    BIOSSetCurPos( (5 << 8) | 20, 0 );
    printf( "Hello world\n" );
}
```

To see how the code generator set up the register values for the in-line code, let us take a look at the disassembled code.

```
   BIOSSetCurPos( (5 << 8) | 20, 0 );
0008  ba 14 05                          mov     dx,0514H
000b  30 ff                            xor     bh,bh
000d  55                               push    bp
000e  b4 02                            mov     ah,02H
0010  cd 10                            int     10H
0012  5d                               pop     bp
```

As we expected, the result of the expression for the row and column is placed in the DX register and the page number is placed in the BH register. The remaining instructions are our in-line code sequence.

Although our examples have been simple, you should be able to generalize them to your situation.

To review, the "parm", "value" and "modify" attributes are used to:

1. convey information to the code generator about the way data values are to be placed in registers in preparation for the code burst (parm),

2. convey information to the code generator about the result, if any, from the code burst (value), and

3. convey information to the code generator about any side effects to the registers after the code burst has executed (modify). It is important to let the code generator

know all of the side effects on registers when the in-line code is executed; otherwise it assumes that all registers, other than those used for parameters, are preserved. In our examples, we chose to push/pop some of the registers that are modified by the code burst.

12.2 Labels in In-line Assembly Code

Labels can be used in in-line assembly code. Here is an example.

Example:

```
extern void _disable_video( unsigned );
#pragma aux _disable_video =    \
 "again: in al,dx"              \
        "test al,8"             \
        "jz again"              \
        "mov dx,03c0h"          \
        "mov al,11h"            \
        "out dx,al"             \
        "mov al,0"              \
        "out dx,al"             \
        parm [dx]               \
        modify [al dx];
```

12.3 Variables in In-line Assembly Code

To finish our discussion, we provide examples that illustrate the use of variables in the in-line assembly code. The following example illustrates the use of static variable references in the auxiliary pragma.

Example:

```
#include <stdio.h>

static short         _rowcol;
static unsigned char _page;
```

```
extern void BIOSSetCurPos( void );
#pragma aux BIOSSetCurPos =     \
        "mov  dx,_rowcol"       \
        "mov  bh,_page"         \
        "push bp"               \
        "mov ah,2"              \
        "int 10h"               \
        "pop bp"                \
        modify [ah bx dx];


void main()
{
    _rowcol = (5 << 8) | 20;
    _page = 0;
    BIOSSetCurPos();
    printf( "Hello world\n" );
}
```

The only rule to follow here is that the auxiliary pragma must be defined after the variables are defined. The in-line assembler is passed information regarding the sizes of variables so they must be defined first.

If we look at a fragment of the disassembled code, we can see the result.

```
   _rowcol = (5 << 8) | 20;
0008  c7 06 00 00 14 05                mov     word ptr __rowcol,0514H

   _page = 0;
000e  c6 06 00 00 00                   mov     byte ptr __page,00H

   BIOSSetCurPos();
0013  8b 16 00 00                      mov     dx,__rowcol
0017  8a 3e 00 00                      mov     bh,__page
001b  55                               push    bp
001c  b4 02                            mov     ah,02H
001e  cd 10                            int     10H
0020  5d                               pop     bp
```

The following example illustrates the use of automatic variable references in the auxiliary pragma. Again, the auxiliary pragma must be defined after the variables are defined so the pragma is placed in-line with the function.

Example:

```
#include <stdio.h>

void main()
{
    short         _rowcol;
    unsigned char _page;

    extern void BIOSSetCurPos( void );
#   pragma aux  BIOSSetCurPos = \
        "mov  dx,_rowcol"       \
        "mov  bh,_page"         \
        "push bp"               \
        "mov ah,2"              \
        "int 10h"               \
        "pop bp"                \
        modify [ah bx dx];

    _rowcol = (5 << 8) | 20;
    _page = 0;
    BIOSSetCurPos();
    printf( "Hello world\n" );
}
```

If we look at a fragment of the disassembled code, we can see the result.

```
   _rowcol = (5 << 8) | 20;
000e  c7 46 fc 14 05                    mov     word ptr -4H[bp],0514H

   _page = 0;
0013  c6 46 fe 00                       mov     byte ptr -2H[bp],00H

   BIOSSetCurPos();
0017  8b 96 fc ff                       mov     dx,-4H[bp]
001b  8a be fe ff                       mov     bh,-2H[bp]
001f  55                                push    bp
0020  b4 02                             mov     ah,02H
0022  cd 10                             int     10H
0024  5d                                pop     bp
```

You should try to avoid references to automatic variables as illustrated by this last example. Referencing automatic variables in this manner causes them to be marked as volatile and the optimizer will not be able to do a good job of optimizing references to these variables.

314 Variables in In-line Assembly Code

12.4 In-line Assembly Language using _asm

There is an alternative to Watcom's auxiliary pragma method for creating in-line assembly code. You can use one of the ***_asm*** or ***__asm*** keywords to imbed assembly code into the generated code. The following is a revised example of the cursor positioning example introduced above.

Example:

```
#include <stdio.h>

void main()
{
    unsigned short _rowcol;
    unsigned char _page;

    _rowcol = (5 << 8) | 20;
    _page = 0;
    _asm {
        mov     dx,_rowcol
        mov     bh,_page
        push    bp
        mov     ah,2
        int     10h
        pop     bp
    };
    printf( "Hello world\n" );
}
```

The assembly language sequence can reference program variables to retrieve or store results. There are a few incompatibilities between Microsoft and Watcom implementation of this directive.

__LOCAL_SIZE is not supported by Watcom C/C++. This is illustrated in the following example.

Example:

```
void main()
{
    int i;
    int j;

    _asm {
        push    bp
        mov     bp,sp
        sub     sp,__LOCAL_SIZE
    };
}
```

structure references are not supported by Watcom C/C++. This is illustrated in the following example.

Example:

```
#include <stdio.h>

struct rowcol {
    unsigned char col;
    unsigned char row;
};

void main()
{
    struct rowcol _pos;
    unsigned char _page;

    _pos.row = 5;
    _pos.col = 20;
    _page = 0;
    _asm {
        mov     dl,_pos.col
        mov     dh,_pos.row
        mov     bh,_page
        push    bp
        mov     ah,2
        int     10h
        pop     bp
    };
    printf( "Hello world\n" );
}
```

12.5 In-line Assembly Directives and Opcodes

It is not the intention of this chapter to describe assembly-language programming in any detail. You should consult a book that deals with this topic. However, we present a list of the directives, opcodes and register names that are recognized by the assembler built into the compiler's auxiliary pragma processor.

```
.8086     .186      .286      .286c     .286p     .386
.386p     .486      .486p     .586      .586p     .8087
.287      .387      aaa       aad       aam       aas
adc       add       ah        al        and       arpl
ax        bh        bl        bound     bp        bsf
bsr       bswap     bt        btc       btr       bts
bx        byte      call      callf     cbw       cdq
ch        cl        clc       cld       cli       clts
cmc       cmp       cmps      cmpsb     cmpsd     cmpsw
cmpxchg   cmpxchg8b cpuid     cr0       cr2       cr3
cr4       cs        cwd       cwde      cx        daa
das       db        dd        dec       df        dh
di        div       dl        dp        dr0       dr1
dr2       dr3       dr6       dr7       ds        dup
dw        dword     dx        eax       ebp       ebx
ecx       edi       edx       enter     es        esi
esp       f2xm1     fabs      fadd      faddp     far
fbld      fbstp     fchs      fclex     fcom      fcomp
fcompp    fcos      fdecstp   fdisi     fdiv      fdivp
fdivr     fdivrp    feni      ffree     fiadd     ficom
ficomp    fidiv     fidivr    fild      fimul     fincstp
finit     fist      fistp     fisub     fisubr    fld
fld1      fldcw     fldenv    fldenvd   fldenvw   fldl2e
fldl2t    fldlg2    fldln2    fldpi     fldz      fmul
fmulp     fnclex    fndisi    fneni     fninit    fnop
fnrstor   fnrstord  fnrstorw  fnsave    fnsaved   fnsavew
fnstcw    fnstenv   fnstenvd  fnstenvw  fnstsw    fpatan
fprem     fprem1    fptan     frndint   frstor    frstord
frstorw   fs        fsave     fsaved    fsavew    fscale
fsetpm    fsin      fsincos   fsqrt     fst       fstcw
fstenv    fstenvd   fstenvw   fstp      fstsw     fsub
fsubp     fsubr     fsubrp    ftst      fucom     fucomp
fucompp   fwait     fword     fxam      fxch      fxtract
fyl2x     fyl2xp1   gs        hlt       idiv      imul
in        inc       ins       insb      insd      insw
int       into      invd      invlpg    iret      iretd
```

```
ja        jae       jb        jbe       jc        jcxz
je        jecxz     jg        jge       jl        jle
jmp       jmpf      jna       jnae      jnb       jnbe
jnc       jne       jng       jnge      jnl       jnle
jno       jnp       jns       jnz       jo        jp
jpe       jpo       js        jz        lahf      lar
lds       lea       leave     les       lfs       lgdt
lgs       lidt      lldt      lmsw      lock      lods
lodsb     lodsd     lodsw     loop      loope     loopne
loopnz    loopz     lsl       lss       ltr       mov
movs      movsb     movsd     movsw     movsx     movzx
mul       near      neg       no87      nop       not
offset    or        out       outs      outsb     outsd
outsw     pop       popa      popad     popf      popfd
ptr       push      pusha     pushad    pushf     pushfd
pword     qword     rcl       rcr       rdmsr     rdtsc
rep       repe      repne     repnz     repz      ret
retf      retn      rol       ror       rsm       sahf
sal       sar       sbb       scas      scasb     scasd
scasw     seg       seta      setae     setb      setbe
setc      sete      setg      setge     setl      setle
setna     setnae    setnb     setnbe    setnc     setne
setng     setnge    setnl     setnle    setno     setnp
setns     setnz     seto      setp      setpe     setpo
sets      setz      sgdt      shl       shld      short
shr       shrd      si        sidt      sldt      smsw
sp        ss        st        stc       std       sti
stos      stosb     stosd     stosw     str       sub
tbyte     test      tr3       tr4       tr5       tr6
tr7       verr      verw      wait      wbinvd    word
wrmsr     xadd      xchg      xlat      xlatb     xor
```

A separate assembler is also included with this product and is described in the ***Watcom C/C++ Tools User's Guide***

Structured Exception Handling in C

13 Structured Exception Handling

Microsoft-style Structured Exception Handling (SEH) is supported by the Watcom C compiler only. MS SEH is supported under the Win32, Win32s and OS/2 platforms. You should not confuse SEH with C++ exception handling. The Watcom C++ compiler supports the standard C++ syntax for exception handling.

The following sections introduce some of the aspects of SEH. For a good description of SEH, please refer to ***Advanced Windows NT*** by Jeffrey Richter (Microsoft Press, 1994). You may also wish to read the article "Clearer, More Comprehensive Error Processing with Win32 Structured Exception Handling" by Kevin Goodman in the January, 1994 issue of Microsoft Systems Journal.

13.1 Termination Handlers

We begin our look at SEH with a simple model. In this model, there are two blocks of code — the "guarded" block and the "termination" block. The termination code is guaranteed to be executed regardless of how the "guarded" block of code is exited (including execution of any "return" statement).

```
_try {
    /* guarded code */
    .
    .
    .
}
_finally {
    /* termination handler */
    .
    .
    .
}
```

The ***_finally*** block of code is guaranteed to be executed no matter how the guarded block is exited (***break, continue, return, goto,*** or ***longjmp()***). Exceptions to this are calls to ***abort(), exit()*** or ***_exit()*** which terminate the execution of the process.

There can be no intervening code between *try* and *finally* blocks.

The following is a contrived example of the use of ***_try*** and ***_finally.***

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <excpt.h>

int docopy( char *in, char *out )
{
  FILE        *in_file = NULL;
  FILE        *out_file = NULL;
  char        buffer[256];

  _try {
    in_file = fopen( in, "r" );
    if( in_file == NULL ) return( EXIT_FAILURE );
    out_file = fopen( out, "w" );
    if( out_file == NULL ) return( EXIT_FAILURE );

    while( fgets((char *)buffer, 255, in_file) != NULL ) {
      fputs( (char *)buffer, out_file );
    }
  }
  _finally {
    if( in_file != NULL ) {
      printf( "Closing input file\n" );
      fclose( in_file );
    }
    if( out_file != NULL ) {
      printf( "Closing output file\n" );
      fclose( out_file );
    }
    printf( "End of processing\n" );
  }
  return( EXIT_SUCCESS );
}

void main( int argc, char **argv )
{
  if( argc < 3 ) {
    printf( "Usage: mv [in_filename] [out_filename]\n" );
    exit( EXIT_FAILURE );
  }
  exit( docopy( argv[1], argv[2] ) );
}
```

The *try* block ignores the messy details of what to do when either one of the input or output files cannot be opened. It simply tests whether a file can be opened and quits if it cannot. The *finally* block ensures that the files are closed if they were opened, releasing the resources associated with open files. This simple example could have been written in C without the use of SEH.

There are two ways to enter the *finally* block. One way is to exit the *try* block using a statement like ***return.*** The other way is to fall through the end of the *try* block and into the *finally* block (the normal execution flow for this program). Any code following the *finally* block is only executed in the second case. You can think of the *finally* block as a special function that is invoked whenever an exit (other than falling out the bottom) is attempted from a corresponding *try* block.

More formally stated, a local unwind occurs when the system executes the contents of a *finally* block because of the premature exit of code in a *try* block.

> ***Note:*** Kevin Goodman describes "unwinds" in his article. "There are two types of unwinds: global and local. A global unwind occurs when there are nested functions and an exception takes place. A local unwind takes place when there are multiple handlers within one function. Unwinding means that the stack is going to be clean by the time your handler's code gets executed."

The *try/finally* structure is a rejection mechanism which is useful when a set of statements is to be conditionally chosen for execution, but not all of the conditions required to make the selection are available beforehand. It is an extension to the C language. You start out with the assumption that a certain task can be accomplished. You then introduce statements into the code that test your hypothesis. The *try* block consists of the code that you assume, under normal conditions, will succeed. Statements like *if ... return* can be used as tests. Execution begins with the statements in the *try* block. If a condition is detected which indicates that the assumption of a normal state of affairs is wrong, a ***return*** statement may be executed to cause control to be passed to the statements in the *finally* block. If the *try* block completes execution without executing a ***return*** statement (i.e., all statements are executed up to the final brace), then control is passed to the first statement following the *try* block (i.e., the first statement in the *finally* block).

In the following example, two sets of codes and letters are read in and some simple sequence checking is performed. If a sequence error is detected, an error message is printed and processing terminates; otherwise the numbers are processed and another pair of numbers is read.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <excpt.h>

void main( int argc, char **argv )
{
  read_file( fopen( argv[1], "r" ) );
}

void read_file( FILE *input )
{
  int         line = 0;
  char        buffer[256];
  char        icode;
  char        x, y;

  if( input == NULL ) {
    printf( "Unable to open file\n" );
    return;
  }

  _try {
    for(;;) {
      line++;
      if( fgets( buffer, 255, input ) == NULL ) break;
      icode = buffer[0];
      if( icode != '1' ) return;
      x = buffer[1];
      line++;
      if( fgets( buffer, 255, input ) == NULL ) return;
      icode = buffer[0];
      if( icode != '2' ) return;
      y = buffer[1];
      process( x, y );
    }
    printf( "Processing complete\n" );
    fclose( input );
    input = NULL;
  }

  _finally {
    if( input != NULL ) {
      printf( "Invalid sequence: line = %d\n", line );
      fclose( input );
    }
  }
}
```

```
void process( char x, char y )
{
    printf( "processing pair %c,%c\n", x, y );
}
```

The above example attempts to read a code and letter. If an end of file occurs then the loop is terminated by the ***break*** statement.

If the code is not 1 then we did not get what we expected and an error condition has arisen. Control is passed to the first statement in the *finally* block by the ***return*** statement. An error message is printed and the open file is closed.

If the code is 1 then a second code and number are read. If an end of file occurs then we are missing a complete set of data and an error condition has arisen. Control is passed to the first statement in the *finally* block by the ***return*** statement. An error message is printed and the open file is closed.

Similarly if the expected code is not 2 an error condition has arisen. The same error handling procedure occurs.

If the second code is 2, the values of variables `x` and `y` are processed (printed). The ***for*** loop is repeated again.

The above example illustrates the point that all the information required to test an assumption (that the file contains valid pairs of data) is not available from the start. We write our code with the assumption that the data values are correct (our hypothesis) and then test the assumption at various points in the algorithm. If any of the tests fail, we reject the hypothesis.

Consider the following example. What values are printed by the program?

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <excpt.h>

void main( int argc, char **argv )
{
  int ctr = 0;
```

```
  while( ctr < 10 ) {
    printf( "%d\n", ctr );
    _try {
      if( ctr == 2 ) continue;
      if( ctr == 3 ) break;
    }
    _finally {
      ctr++;
    }

    ctr++;
  }
  printf( "%d\n", ctr );
}
```

At the top of the loop, the value of `ctr` is 0. The next time we reach the top of the loop, the value of `ctr` is 2 (having been incremented twice, once by the *finally* block and once at the bottom of the loop). When `ctr` has the value 2, the ***continue*** statement will cause the *finally* block to be executed (resulting in `ctr` being incremented to 3), after which execution continues at the top of the ***while*** loop. When `ctr` has the value 3, the ***break*** statement will cause the *finally* block to be executed (resulting in `ctr` being incremented to 4), after which execution continues after the ***while*** loop. Thus the output is:

```
0
2
3
4
```

The point of this exercise was that after the *finally* block is executed, the normal flow of execution is resumed at the ***break, continue, return,*** etc. statement and the normal behaviour for that statement occurs. It is as if the compiler had inserted a function call just before the statement that exits the *try* block.

```
_try {
  if( ctr == 2 ) invoke_finally_block() continue;
  if( ctr == 3 ) invoke_finally_block() break;
}
```

There is some overhead associated with local unwinds such as that incurred by the use of ***break, continue, return,*** etc. To avoid this overhead, a new transfer keyword called ***_leave*** can be used. The use of this keyword causes a jump to the end of the *try block.* Consider the following modified version of an earlier example.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <excpt.h>

void main( int argc, char **argv )
{
  read_file( fopen( argv[1], "r" ) );
}

void read_file( FILE *input )
{
  int         line = 0;
  char        buffer[256];
  char        icode;
  char        x, y;

  if( input == NULL ) {
    printf( "Unable to open file\n" );
    return;
  }

  _try {
    for(;;) {
      line++;
      if( fgets( buffer, 255, input ) == NULL ) break;
      icode = buffer[0];
      if( icode != '1' ) _leave;
      x = buffer[1];
      line++;
      if( fgets( buffer, 255, input ) == NULL ) _leave;
      icode = buffer[0];
      if( icode != '2' ) _leave;
      y = buffer[1];
      process( x, y );
    }
    printf( "Processing complete\n" );
    fclose( input );
    input = NULL;
  }

  _finally {
    if( input != NULL ) {
      printf( "Invalid sequence: line = %d\n", line );
      fclose( input );
    }
  }
}
```

```
void process( char x, char y )
{
  printf( "processing pair %c,%c\n", x, y );
}
```

There are two ways to enter the *finally* block. One way is caused by unwinds — either local (by the use of ***break, continue, return,*** or ***goto***) or global (more on global unwinds later). The other way is through the normal flow of execution (i.e., simply by falling through the bottom of the *try* block). There is a function called ***AbnormalTermination*** that can be used to determine which of these two methods was used to enter the *finally* block. If the function returns TRUE (1) then the *finally* block was entered using the first method; if the function returns FALSE (0) then the *finally* block was entered using the second method. This information may be useful in some circumstances. For example, you may wish to avoid executing any code in a *finally* block if the block was entered through the normal flow of execution.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <excpt.h>

void main( int argc, char **argv )
{
  read_file( fopen( argv[1], "r" ) );
}

void read_file( FILE *input )
{
  int         line = 0;
  char        buffer[256];
  char        icode;
  char        x, y;

  if( input == NULL ) {
    printf( "Unable to open file\n" );
    return;
  }
```

```
  _try {
    for(;;) {
      line++;
      if( fgets( buffer, 255, input ) == NULL ) break;
      icode = buffer[0];
      if( icode != '1' ) return;
      x = buffer[1];
      line++;
      if( fgets( buffer, 255, input ) == NULL ) return;
      icode = buffer[0];
      if( icode != '2' ) return;
      y = buffer[1];
      process( x, y );
    }
    printf( "Processing complete\n" );
  }

  _finally {
    if( AbnormalTermination() )
        printf( "Invalid sequence: line = %d\n", line );
    fclose( input );
  }
}

void process( char x, char y )
{
  printf( "processing pair %c,%c\n", x, y );
}
```

In the above example, we reverted back to the use of the ***return*** statement since the execution of a ***_leave*** statement is considered part of the normal flow of execution and is not considered an "abnormal termination" of the *try* block. Note that since it is not possible to determine whether the *finally* block is executing as the result of a local or global unwind, it may not be appropriate to use the ***AbnormalTermination*** function as a way to determine what has gone on. However, in our simple example, we expect that nothing could go wrong in the "processing" routine.

13.2 Exception Filters and Exception Handlers

We would all like to create flawless software but situations arise for which we did not plan. An event that we did not expect which causes the software to cease to function properly is called an exception. The computer can generate a hardware exception when the software attempts to execute an illegal instruction. We can force this quite easily in C by dereferencing a NULL pointer as shown in the following sample fragment of code.

Example:

```
char *nullp = NULL;

*nullp = '\1';
```

We can also generate software exceptions from software by calling a special function for this purpose. We will look at software exceptions in more detail later on.

Given that exceptions are generally very difficult to avoid in large software projects, we can acknowledge that they are a fact of life and prepare for them. A mechanism similar to *try/finally* has been devised that makes it possible to gain control when an exception occurs and to execute procedures to handle the situation.

The exception handling mechanism involves the pairing up of a ***_try*** block with an ***_except*** block. This is illustrated in the following example.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <excpt.h>

void main( int argc, char **argv )
{
  char *nullp = NULL;

  printf( "Attempting illegal memory reference.\n" );
  _try {
    *nullp = '\1';
  }
  _except (EXCEPTION_EXECUTE_HANDLER) {
    printf( "Oh no! We had an exception!\n" );
  }
  printf( "We recovered fine...\n" );
}
```

In this example, any exception that occurs while executing "inside" the *try* block will cause the *except* block to execute. Unlike the *finally* block, execution of the *except* block occurs

only when an exception is generated and only when the expression after the ***_except*** keyword evaluates to `EXCEPTION_EXECUTE_HANDLER.` The expression can be quite complex and can involve the execution of a function that returns one of the permissible values. The expression is called the exception "filter" since it determines whether or not the exception is to be handled by the *except* block. The permissible result values for the exception filer are:

EXCEPTION_EXECUTE_HANDLER
meaning "I will handle the exception".

EXCEPTION_CONTINUE_EXECUTION
meaning "I want to resume execution at the point where the exception was generated".

EXCEPTION_CONTINUE_SEARCH
meaning "I do not want to handle the exception so continue looking down the *try/except* chain until you find an exception handler that does want to handle the exception".

13.3 Resuming Execution After an Exception

Why would you want to resume execution of the instruction that caused the exception? Since the exception filter can involve a function call, that function can attempt to correct the problem. For example, if it is determined that the exception has occurred because of the NULL pointer dereference, the function could modify the pointer so that it is no longer NULL.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <excpt.h>

char *NullP = NULL;

int filter()
{
  if( NullP == NULL ) {
    NullP == malloc( 20 );
    return( EXCEPTION_CONTINUE_EXECUTION )
  }
  return( EXCEPTION_EXECUTE_HANDLER )
}
```

```
void main( int argc, char **argv )
{

  printf( "Attempting illegal memory reference.\n" );
  _try {
    *NullP = '\1';
  }

  _except (filter()) {
    printf( "Oh no! We had an exception!\n" );
  }
  printf( "We recovered fine...\n" );
}
```

Unfortunately, this is does not solve the problem. Understanding why it does not involves looking at the sequence of computer instructions that is generated for the expression in question.

```
        *NullP = '\1';
            mov     eax,dword ptr _NullP
            mov     byte ptr [eax],01H
```

The exception is caused by the second instruction which contains a pointer to the referenced memory location (i.e., 0) in register EAX. This is the instruction that will be repeated when the filter returns `EXCEPTION_CONTINUE_EXECUTION.` Since EAX did not get changed by our fix, the exception will reoccur. Fortunately, `NullP` is changed and this prevents our program from looping forever. The moral here is that there are very few instances where you can correct "on the fly" a problem that is causing an exception to occur. Certainly, any attempt to do so must involve a careful inspection of the computer instruction sequence that is generated by the compiler (and this sequence usually varies with the selection of compiler optimization options). The best solution is to add some more code to detect the problem before the exception occurs.

13.4 Mixing and Matching _try/_finally and _try/_except

Where things really get interesting is in the interaction between *try/finally* blocks and *try/except* blocks. These blocks can be nested within each other. In an earlier part of the discussion, we talked about global unwinds and how they can be caused by exceptions being generated in nested function calls. All of this should become clear after studying the following example.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <excpt.h>

func_level4()
{
  char *nullp = NULL;

  printf( "Attempting illegal memory reference\n" );
  _try {
    *nullp = '\1';
  }
  _finally {
    if( AbnormalTermination() )
      printf( "Unwind in func_level4\n" );
  }
  printf( "Normal return from func_level4\n" );
}
func_level3()
{
  _try {
    func_level4();
  }
  _finally {
    if( AbnormalTermination() )
      printf( "Unwind in func_level3\n" );
  }
  printf( "Normal return from func_level3\n" );
}
func_level2()
{
  _try {
    _try {
      func_level3();
    }
    _except (EXCEPTION_CONTINUE_SEARCH) {
      printf( "Exception never handled in func_level2\n" );
    }
  }
  _finally {
    if( AbnormalTermination() )
      printf( "Unwind in func_level2\n" );
  }
  printf( "Normal return from func_level2\n" );
}
func_level1()
{
  _try {
    func_level2();
  }
  _finally {
    if( AbnormalTermination() )
      printf( "Unwind in func_level1\n" );
  }
  printf( "Normal return from func_level1\n" );
}
```

```
func_level0()
{
  _try {
    _try {
      func_level1();
    }
    _except (EXCEPTION_EXECUTE_HANDLER) {
      printf( "Exception handled in func_level0\n" );
    }
  }
  _finally {
    if( AbnormalTermination() )
      printf( "Unwind in func_level0\n" );
  }
  printf( "Normal return from func_level0\n" );
}
void main( int argc, char **argv )
{
  _try {
    _try {
      func_level0();
    }
    _except (EXCEPTION_EXECUTE_HANDLER) {
      printf( "Exception handled in main\n" );
    }
  }
  _finally {
    if( AbnormalTermination() )
      printf( "Unwind in main\n" );
  }
  printf( "Normal return from main\n" );
}
```

In this example,

1. `main` calls `func_level0`
2. `func_level0` calls `func_level1`
3. `func_level1` calls `func_level2`
4. `func_level2` calls `func_level3`
5. `func_level3` calls `func_level4`

It is in `func_level4` where the exception occurs. The run-time system traps the exception and performs a search of the active *try* blocks looking for one that is paired up with an *except* block.

When it finds one, the filter is executed and, if the result is `EXCEPTION_EXECUTE_HANDLER,` then the *except* block is executed after performing a global unwind.

If the result is EXCEPTION_CONTINUE_EXECUTION, the run-time system resumes execution at the instruction that caused the exception.

If the result is EXCEPTION_CONTINUE_SEARCH, the run-time system continues its search for an *except* block with a filter that returns one of the other possible values. If it does not find any exception handler that is prepared to handle the exception, the application will be terminated with the appropriate exception notification.

Let us look at the result of executing the example program. The following messages are printed.

```
Attempting illegal memory reference
Unwind in func_level4
Unwind in func_level3
Unwind in func_level2
Unwind in func_level1
Exception handled in func_level0
Normal return from func_level0
Normal return from main
```

The run-time system searched down the *try/except* chain until it got to `func_level0` which had an *except* filter that evaluated to `EXCEPTION_EXECUTE_HANDLER.` It then performed a global unwind in which the *try/finally* blocks of `func_level4, func_level3, func_level2,` and `func_level1` were executed. After this, the exception handler in `func_level0` did its thing and execution resumed in `func_level0` which returned back to `main` which returned to the run-time system for normal program termination. Note the use of the built-in ***AbnormalTermination*** function in the *finally* blocks of each function.

This sequence of events permits each function to do any cleaning up that it deems necessary before it is wiped off the execution stack.

13.5 Refining Exception Handling

The decision to handle an exception must be weighed carefully. It is not necessarily a desirable thing for an exception handler to handle all exceptions. In the previous example, the expression in the exception filter in `func_level0` always evaluates to `EXCEPTION_EXECUTE_HANDLER` which means it will snag every exception that comes its way. There may be other exception handlers further on down the chain that are better equipped to handle certain types of exceptions. There is a way to determine the exact type of exception using the built-in `GetExceptionCode()` function. It may be called only from within the exception handler filter expression or within the exception handler block. Here is a description of the possible return values from the `GetExceptionCode()` function.

Value	*Meaning*
EXCEPTION_ACCESS_VIOLATION	The thread tried to read from or write to a virtual address for which it does not have the appropriate access.
EXCEPTION_BREAKPOINT	A breakpoint was encountered.
EXCEPTION_DATATYPE_MISALIGNMENT	The thread tried to read or write data that is misaligned on hardware that does not provide alignment. For example, 16-bit values must be aligned on 2-byte boundaries; 32-bit values on 4-byte boundaries, and so on.
EXCEPTION_SINGLE_STEP	A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	The thread tried to access an array element that is out of bounds and the underlying hardware supports bounds checking.
EXCEPTION_FLT_DENORMAL_OPERAND	One of the operands in a floating-point operation is denormal. A denormal value is one that is too small to represent as a standard floating-point value.
EXCEPTION_FLT_DIVIDE_BY_ZERO	The thread tried to divide a floating-point value by a floating-point divisor of zero.
EXCEPTION_FLT_INEXACT_RESULT	The result of a floating-point operation cannot be represented exactly as a decimal fraction.
EXCEPTION_FLT_INVALID_OPERATION	This exception represents any floating-point exception not included in this list.
EXCEPTION_FLT_OVERFLOW	The exponent of a floating-point operation is greater than the magnitude allowed by the corresponding type.

EXCEPTION_FLT_STACK_CHECK
: The stack overflowed or underflowed as the result of a floating-point operation.

EXCEPTION_FLT_UNDERFLOW
: The exponent of a floating-point operation is less than the magnitude allowed by the corresponding type.

EXCEPTION_INT_DIVIDE_BY_ZERO
: The thread tried to divide an integer value by an integer divisor of zero.

EXCEPTION_INT_OVERFLOW
: The result of an integer operation caused a carry out of the most significant bit of the result.

EXCEPTION_PRIV_INSTRUCTION
: The thread tried to execute an instruction whose operation is not allowed in the current machine mode.

EXCEPTION_NONCONTINUABLE_EXCEPTION
: The thread tried to continue execution after a non-continuable exception occurred.

These constants are defined by including `WINDOWS.H` in the source code.

The following example is a refinement of the `func_level1()` function in our previous example.

Example:

```
#include <windows.h>

func_level0()
{
  _try {
    _try {
      func_level1();
    }
    _except (
        (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION)
        ? EXCEPTION_EXECUTE_HANDLER
        : EXCEPTION_CONTINUE_SEARCH
        ) {
      printf( "Exception handled in func_level0\n" );
    }
  }
  _finally {
    if( AbnormalTermination() )
      printf( "Unwind in func_level0\n" );
  }
  printf( "Normal return from func_level0\n" );
}
```

In this version, only an "access violation" will be handled by the exception handler in the `func_level0()` function. All other types of exceptions will be passed on to `main` (which can also be modified to be somewhat more selective about the types of exceptions it should handle).

More information on the exception that has occurred can be obtained by the use of the `GetExceptionInformation()` function. The use of this function is also restricted. It can be called only from within the filter expression of an exception handler. However, the return value of `GetExceptionInformation()` can be passed as a parameter to a filter function. This is illustrated in the following example.

Example:

```
int GetCode( LPEXCEPTION_POINTERS exceptptrs )
{
    return (exceptptrs->ExceptionRecord->ExceptionCode );
}

func_level0()
{
  _try {
    _try {
      func_level1();
    }
    _except (
        (GetCode( GetExceptionInformation() )
            == EXCEPTION_ACCESS_VIOLATION)
        ? EXCEPTION_EXECUTE_HANDLER
        : EXCEPTION_CONTINUE_SEARCH
        ) {
      printf( "Exception handled in func_level0\n" );
    }
  }
  _finally {
    if( AbnormalTermination() )
      printf( "Unwind in func_level0\n" );
  }
  printf( "Normal return from func_level0\n" );
}
```

The return value of `GetExceptionInformation()` is a pointer to an `EXCEPTION_POINTERS` structure that contains pointers to two other structures: an `EXCEPTION_RECORD` structure containing a description of the exception, and a `CONTEXT` structure containing the machine-state information. The filter function can make a copy of the structures if a more permanent copy is desired. Check your Win32 SDK documentation for more information on these structures.

13.6 Throwing Your Own Exceptions

You can use the same exception handling mechanisms to deal with software exceptions raised by your application. The `RaiseException()` function can be used to throw your own application-defined exceptions. The first argument to this function is the exception code. It would be wise to define your exception codes so that they do not collide with system defined ones. The following example shows how to throw an exception.

Example:

```
#define MY_EXCEPTION ( (DWORD) 123L )

    RaiseException( MY_EXCEPTION,
                    EXCEPTION_NONCONTINUABLE,
                    0, NULL );
```

In this example, the `GetExceptionCode()` function, when used in an exception handler filter expression or in the body of an exception handler, would return the value 123.

See the Win32 SDK documentation for more information on the arguments to the `RaiseException()` function.

Embedded Systems

14 Creating ROM-based Applications

14.1 Introduction

This chapter provides information for developers who wish to write applications to be placed in read-only memory (ROM).

14.2 ROMable Functions

The following functions in the Watcom C/C++ library are not dependent on any operating system. Therefore they can be used for embedded applications. The math functions are listed here because they are ROMable, however you must supply a different `_matherr` function if you are not running in the DOS, OS/2 or Windows NT environment.

abs	acos	alloca
asctime	asin	atan
atan2	atexit	atof
atoi	atol	bsearch
cabs	ceil	_chain_intr
_clear87	_control87	cos
cosh	difftime	_disable
div	_enable	exp
fabs	floor	_fmemccpy
_fmemchr	_fmemcmp	_fmemcpy
_fmemicmp	_fmemmove	_fmemset
fmod	FP_OFF	FP_SEG
_fpreset	frexp	_fstrcat
_fstrchr	_fstrcmp	_fstrcpy
_fstrcspn	_fstricmp	_fstrlen
_fstrlwr	_fstrncat	_fstrncmp
_fstrncpy	_fstrnicmp	_fstrnset
_fstrpbrk	_fstrrchr	_fstrrev
_fstrset	_fstrspn	_fstrstr
_fstrtok	_fstrupr	gmtime
hypot	inp	inpw
int86 (1)	int86x (1)	int386 (2)
int386x (2)	intr	isalnum
isalpha	isascii	iscntrl

```
isdigit          isgraph          islower
isprint          ispunct          isspace
isupper          isxdigit         itoa
j0               j1               jn
labs             ldexp            ldiv
lfind            localeconv       log
log10            longjmp          _lrotl
_lrotr           lsearch          ltoa
matherr          mblen            mbstowcs
mbtowc           memccpy          memchr
memcmp           memcpy           memicmp
memmove          memset           MK_FP
modf             movedata         offsetof
outp             outpw            pow
qsort            rand             _rotl
_rotr            segread          setjmp
setlocale        sin              sinh
sprintf          sqrt             srand
sscanf           stackavail       _status87
strcat           strchr           strcmp
strcmpi          strcoll          strcpy
strcspn          strdup           strerror
stricmp          strlen           strlwr
strncat          strncmp          strncpy
strnicmp         strnset          strpbrk
strrchr          strrev           strset
strspn           strstr           strtod
strtok           strtol           strtoul
strupr           strxfrm          swab
tan              tanh             tolower
toupper          ultoa            utoa
va_arg           va_end           va_start
vsprintf         vsscanf          wcstombs
wctomb           y0               y1
yn
```

* (1) 16-bit libraries
* (2) 32-bit libraries

14.3 System-Dependent Functions

The following functions in the C/C++ library directly or indirectly make use of operating system functions. They cannot be used on systems that are not running on one of the DOS, OS/2 or Windows NT operating systems.

```
abort               access              assert
bdos                _beginthread        _bios_disk
_bios_equiplist     _bios_keybrd        _bios_memsize
_bios_printer       _bios_serialcom     _bios_timeofday
calloc              cgets               chdir
chmod               chsize              clearerr
clock               close               closedir
cprintf             cputs               creat
cscanf              ctime               cwait
delay               _dos_allocmem       _dos_close
_dos_creat          _dos_creatnew       _dos_findfirst
_dos_findnext       _dos_freemem        _dos_getdate
_dos_getdiskfree    _dos_getdrive       _dos_getfileattr
_dos_getftime       _dos_gettime        _dos_getvect
_dos_keep           _dos_open           _dos_read
_dos_setblock       _dos_setdate        _dos_setdrive
_dos_setfileattr    _dos_setftime       _dos_settime
_dos_setvect        _dos_write          dosexterr
dup                 dup2                _endthread
eof                 execl (1)           execle (1)
execlp (1)          execlpe (1)         execv (1)
execve (1)          execvp (1)          execvpe (1)
exit                _exit               fclose
fcloseall           fdopen              feof
ferror              fflush              _ffree
_fheapchk           _fheapgrow (1)      _fheapmin
_fheapset           _fheapshrink        _fheapwalk
fgetc               fgetpos             fgets
filelength          fileno              flushall
_fmalloc            fopen               fprintf
fputc               fputs               fread
_frealloc           free                freopen
fscanf              fseek               fsetpos
fstat               ftell               fwrite
getc                getch               getchar
getche              getcmd              getcwd
getenv              getpid              gets
halloc              _heapchk            _heapgrow
_heapmin            _heapset            _heapshrink
_heapwalk           hfree               intdos
```

```
intdosx             isatty              kbhit
localtime           lock                locking
lseek               _makepath           malloc
mkdir               mktime              _nfree
_nheapchk           _nheapgrow          _nheapmin
_nheapset           _nheapshrink        _nheapwalk
_nmalloc            _nrealloc           nosound
open                opendir             perror
printf              putc                putch
putchar             putenv              puts
raise               read                readdir
realloc             remove              rename
rewind              rmdir               sbrk
scanf               _searchenv          setbuf
setmode             setvbuf             signal
sleep               sopen               sound
spawnl              spawnle             spawnlp
spawnlpe            spawnv              spawnve
spawnvp             spawnvpe            _splitpath
stat                strftime            system
tell                time                tmpfile
tmpnam              tzset               umask
ungetc              ungetch             unlink
unlock              utime               vfprintf
vfscanf             vprintf             vscanf
wait                write
```

* (1) 16-bit libraries

14.4 Modifying the Startup Code

Source files are included in the package for the Watcom C/C++ application start-up (or initialization) sequence. These files are described in the section entitled "The Watcom C/C++ Run-time Initialization Routines" on page 130. The startup code will have to be modified if you are creating a ROMable application or you are not running in a DOS, OS/2, QNX, or Windows environment.

14.5 Choosing the Correct Floating-Point Option

If there will be a math coprocessor chip in your embedded system, then you should compile your application with the "fpi87" option and one of "fp2", "fp3" or "fp5" depending on which math coprocessor chip will be in your embedded system. If there will not be a math coprocessor chip in your embedded system, then you should compile your application with the "fpc" option. You should not use the "fpi" option since that will cause extra code to be linked into your application to decode and emulate the 80x87 instructions contained in your application.

Appendices

A. Use of Environment Variables

In the Watcom C/C++ software development package, a number of environment variables are used. This appendix summarizes their use with a particular component of the package.

A.1 FORCE

The **FORCE** environment variable identifies a file that is to be included as part of the source input stream. This variable is used by Watcom C/C++.

```
SET FORCE=[d:][path]filename[.ext]
```

The specified file is included as if a

```
#include "[d:][path]filename[.ext]"
```

directive were placed at the start of the source file.

Example:

```
C>set force=\watcom\h\common.cnv
C>wcc report
```

The **FORCE** environment variable can be overridden by use of the Watcom C/C++ "fi" option.

A.2 INCLUDE

The **INCLUDE** environment variable describes the location of the C and C++ header files (files with the ".h" filename extension). This variable is used by Watcom C/C++.

```
SET include=[d:][path];[d:][path]...
```

The **INCLUDE** environment string is like the **PATH** string in that you can specify one or more directories separated by semicolons (";").

A.3 LIB

The use of the **WATCOM** environment variable and the Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

The **LIB** environment variable is used to select the libraries that will be used when the application is linked. This variable is used by the Watcom Linker (WLINK.EXE). The **LIB** environment string is like the **PATH** string in that you can specify one or more directories separated by semicolons (";").

If you have the 286 development system, 16-bit applications can be linked for DOS, Microsoft Windows, OS/2, and QNX depending on which libraries are selected. If you have the 386 development system, 32-bit applications can be linked for DOS Extender systems, Microsoft Windows and QNX.

A.4 LIBDOS

The use of the **WATCOM** environment variable and the Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

If you are developing a DOS application, the **LIBDOS** environment variable must include the location of the 16-bit Watcom C/C++ DOS library files (files with the ".lib" filename extension). This variable is used by the Watcom Linker (WLINK.EXE). The default installation directory for the 16-bit Watcom C/C++ DOS libraries is `\WATCOM\LIB286\DOS`. The **LIBDOS** environment variable must also include the location of the 16-bit Watcom C/C++ math library files. The default installation directory for the 16-bit Watcom C/C++ math libraries is `\WATCOM\LIB286`.

Example:

```
C>set libdos=c:\watcom\lib286\dos;c:\watcom\lib286
```

A.5 LIBWIN

The use of the **WATCOM** environment variable and the Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

If you are developing a 16-bit Microsoft Windows application, the **LIBWIN** environment variable must include the location of the 16-bit Watcom C/C++ Windows library files (files with the ".lib" filename extension). This variable is used by the Watcom Linker (WLINK.EXE). If you are developing a 32-bit Microsoft Windows application, see the

description of the **LIBPHAR** environment variable. The default installation directory for the 16-bit Watcom C/C++ Windows libraries is `\WATCOM\LIB286\WIN`. The **LIBWIN** environment variable must also include the location of the 16-bit Watcom C/C++ math library files. The default installation directory for the 16-bit Watcom C/C++ math libraries is `\WATCOM\LIB286`.

Example:

```
C>set libwin=c:\watcom\lib286\win;c:\watcom\lib286
```

A.6 LIBOS2

The use of the **WATCOM** environment variable and the Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

If you are developing an OS/2 application, the **LIBOS2** environment variable must include the location of the 16-bit Watcom C/C++ OS/2 library files (files with the ".lib" filename extension). This variable is used by the Watcom Linker (WLINK.EXE). The default installation directory for the 16-bit Watcom C/C++ OS/2 libraries is `\WATCOM\LIB286\OS2`. The **LIBOS2** environment variable must also include the directory of the OS/2 `DOSCALLS.LIB` file which is usually `\OS2`. The **LIBOS2** environment variable must also include the location of the 16-bit Watcom C/C++ math library files. The default installation directory for the 16-bit Watcom C/C++ math libraries is `\WATCOM\LIB286`.

Example:

```
C>set libos2=c:\watcom\lib286\os2;c:\watcom\lib286;c:\os2
```

A.7 LIBPHAR

The use of the **WATCOM** environment variable and the Watcom Linker "SYSTEM" directive is recommended over the use of this environment variable.

If you are developing a 32-bit Windows or DOS Extender application, the **LIBPHAR** environment variable must include the location of the 32-bit Watcom C/C++ DOS Extender library files or the 32-bit Watcom C/C++ Windows library files (files with the ".lib" filename extension). This variable is used by the Watcom Linker (WLINK.EXE). The default installation directory for the 32-bit Watcom C/C++ DOS Extender libraries is `\WATCOM\LIB386\DOS`. The default installation directory for the 32-bit Watcom C/C++ Windows libraries is `\WATCOM\LIB386\WIN`. The **LIBPHAR** environment variable must also include the location of the 32-bit Watcom C/C++ math library files. The default installation directory for the 32-bit Watcom C/C++ math libraries is `\WATCOM\LIB386`.

Example:

```
C>set libphar=c:\watcom\lib386\dos;c:\watcom\lib386
    or
C>set libphar=c:\watcom\lib386\win;c:\watcom\lib386
```

A.8 NO87

The **NO87** environment variable is checked by the Watcom run-time math libraries that include floating-point emulation support. Normally, these libraries will detect the presence of a numeric data processor (80x87) and use it. If you have a numeric data processor in your system but you wish to test a version of your application that will use floating-point emulation, you can define the **NO87** environment variable. Using the "SET" command, define the environment variable as follows:

```
SET NO87=1
```

Now, when you run your application, the 80x87 will be ignored. To undefine the environment variable, enter the command:

```
SET NO87=
```

A.9 PATH

The **PATH** environment variable is used by DOS "COMMAND.COM" or OS/2 "CMD.EXE" to locate programs.

```
PATH [d:][path];[d:][path]...
```

The **PATH** environment variable should include the disk and directory of the Watcom C/C++ binary program files when using Watcom C/C++ and its related tools.

If your host system is DOS:

The default installation directory for 16-bit Watcom C/C++ and 32-bit Watcom C/C++ DOS binaries is called `\WATCOM\BINW`.

Example:

```
C>path c:\watcom\binw;c:\dos;c:\windows
```

If your host system is OS/2:

The default installation directories for 16-bit Watcom C/C++ and 32-bit Watcom C/C++ OS/2 binaries are called `\WATCOM\BINP` and `\WATCOM\BINW.`

Example:

```
[C:\]path c:\watcom\binp;c:\watcom\binw
```

If your host system is Windows NT:

The default installation directories for 16-bit Watcom C/C++ and 32-bit Watcom C/C++ Windows NT binaries are called `\WATCOM\BINNT` and `\WATCOM\BINW.`

Example:

```
C>path c:\watcom\binnt;c:\watcom\binw
```

The **PATH** environment variable is also used by the following programs in the described manner.

1. Watcom Compile and Link to locate the 16-bit Watcom C/C++ and 32-bit Watcom C/C++ compilers and the Watcom Linker.
2. "WD.EXE" to locate programs and debugger command files.

A.10 TMP

The **TMP** environment variable describes the location (disk and path) for temporary files created by the 16-bit Watcom C/C++ and 32-bit Watcom C/C++ compilers and the Watcom Linker.

```
SET TMP=[d:][path]
```

Normally, Watcom C/C++ will create temporary spill files in the current directory. However, by defining the **TMP** environment variable to be a certain disk and directory, you can tell Watcom C/C++ where to place its temporary files. The same is true of the Watcom Linker temporary file.

Consider the following definition of the **TMP** environment variable.

Example:

```
C>set tmp=d:\watcom\tmp
```

The Watcom C/C++ compiler and Watcom Linker will create its temporary files in `D:\WATCOM\TMP.`

A.11 WATCOM

In order for the Watcom Linker to locate the 16-bit Watcom C/C++ and 32-bit Watcom C/C++ library files, the **WATCOM** environment variable should be defined. The **WATCOM** environment variable is used to locate the libraries that will be used when the application is linked. The default directory for 16-bit Watcom C/C++ and 32-bit Watcom C/C++ files is "\WATCOM".

Example:

```
C>set watcom=c:\watcom
```

A.12 WCC

The **WCC** environment variable can be used to specify commonly-used options for the 16-bit C compiler.

```
SET WCC=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wcc=/d1 /ox
```

Once the **WCC** environment variable has been defined, those options listed become the default each time the ***WCC*** command is used.

A.13 WCC386

The **WCC386** environment variable can be used to specify commonly-used options for the 32-bit C compiler.

```
SET WCC386=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wcc386=/d1 /ox
```

Once the **WCC386** environment variable has been defined, those options listed become the default each time the ***WCC386*** command is used.

A.14 WCL

The **WCL** environment variable can be used to specify commonly-used WCL options.

```
SET WCL=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "mm" (compile code for medium memory model), "d1" (include line number debug information in the object file), and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wcl=/mm /d1 /ox
```

Once the **WCL** environment variable has been defined, those options listed become the default each time the WCL command is used.

A.15 WCL386

The **WCL386** environment variable can be used to specify commonly-used WCL386 options.

```
SET WCL386=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "3s" (compile code for stack-based argument passing convention), "d1" (include line number debug information in the object file), and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wcl386=/3s /d1 /ox
```

Once the **WCL386** environment variable has been defined, those options listed become the default each time the WCL386 command is used.

A.16 WCGMEMORY

The **WCGMEMORY** environment variable may be used to request a report of the amount of memory used by the compiler's code generator for its work area.

Example:

```
C>set WCGMEMORY=?
```

When the memory amount is "?" then the code generator will report how much memory was used to generate the code.

It may also be used to instruct the compiler's code generator to allocate a fixed amount of memory for a work area.

Example:

```
C>set WCGMEMORY=128
```

When the memory amount is "nnn" then exactly "nnnK" bytes will be used. In the above example, 128K bytes is requested. If less than "nnnK" is available then the compiler will quit with a fatal error message. If more than "nnnK" is available then only "nnnK" will be used.

There are two reasons why this second feature may be quite useful. In general, the more memory available to the code generator, the more optimal code it will generate. Thus, for two personal computers with different amounts of memory, the code generator may produce different (although correct) object code. If you have a software quality assurance requirement

that the same results (i.e., code) be produced on two different machines then you should use this feature. To generate identical code on two personal computers with different memory configurations, you must ensure that the **WCGMEMORY** environment variable is set identically on both machines.

The second reason where this feature is useful is on virtual memory paging systems (e.g., OS/2) where an unlimited amount of memory can be used by the code generator. If a very large module is being compiled, it may take a very long time to compile it. The code generator will continue to allocate more and more memory and cause an excessive amount of paging. By restricting the amount of memory that the code generator can use, you can reduce the amount of time required to compile a routine.

A.17 WD

The **WD** environment variable can be used to specify commonly-used Watcom Debugger options. This environment variable is not used by the Windows version of the debugger, WDW.

```
SET WD=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "noinvoke" (do not execute the `PROFILE.DBG` file) and "reg=10" (retain up to 10 register sets while tracing).

Example:

```
C>set wd=/noinvoke /reg#10
```

Once the **WD** environment variable has been defined, those options listed become the default each time the WD command is used.

A.18 WDW

The **WDW** environment variable can be used to specify commonly-used Watcom Debugger options. This environment variable is used by the Windows version of the debugger, WDW.

```
SET WDW=/option1 /option2 ...
```

These options are processed before options specified in the WDW prompt dialogue box. The following example defines the default options to be "noinvoke" (do not execute the `PROFILE.DBG` file) and "reg=10" (retain up to 10 register sets while tracing).

Example:

```
C>set wdw=/noinvoke /reg#10
```

Once the **WDW** environment variable has been defined, those options listed become the default each time the WDW command is used.

A.19 WLANG

The **WLANG** environment variable can be used to control which language is used to display diagnostic and program usage messages by various Watcom software tools. The two currently-supported values for this variable are "English" or "Japanese".

```
SET WLANG=English
SET WLANG=Japanese
```

Alternatively, a numeric value of 0 (for English) or 1 (for Japanese) can be specified.

Example:

```
C>set wlang=0
```

By default, Japanese messages are displayed when the current codepage is 932 and English messages are displayed otherwise. Normally, use of the **WLANG** environment variable should not be required.

A.20 WPP

The **WPP** environment variable can be used to specify commonly-used options for the 16-bit C++ compiler.

```
SET WPP=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wpp=/d1 /ox
```

Once the **WPP** environment variable has been defined, those options listed become the default each time the ***WPP*** command is used.

A.21 WPP386

The **WPP386** environment variable can be used to specify commonly-used options for the 32-bit C++ compiler.

```
SET WPP386=/option1 /option2 ...
```

These options are processed before options specified on the command line. The following example defines the default options to be "d1" (include line number debug information in the object file) and "ox" (compile for maximum number of code optimizations).

Example:

```
C>set wpp386=/d1 /ox
```

Once the **WPP386** environment variable has been defined, those options listed become the default each time the ***WPP386*** command is used.

B. Watcom C Diagnostic Messages

The following is a list of all warning and error messages produced by the Watcom C compilers. Diagnostic messages are issued during compilation and execution.

The messages listed in the following sections contain references to `%s`, `%d` and `%u`. They represent strings that are substituted by the Watcom C compilers to make the error message more exact. `%d` and `%u` represent a string of digits; `%s` a string, usually a symbolic name.

Consider the following program, named `ERR.C`, which contains errors.

Example:

```
#include <stdio.h>

void main()
  {
    int i;
    float i;

    i = 383;
    x = 13143.0;
    printf( "Integer value is %d\n", i );
    printf( "Floating-point value is %f\n", x );
  }
```

If we compile the above program, the following messages will appear on the screen.

```
err.c(6): Error! E1034: Symbol 'i' already defined
err.c(9): Error! E1011: Symbol 'x' has not been declared
err.c: 12 lines, included 191, 0 warnings, 2 errors
```

The diagnostic messages consist of the following information:

1. the name of the file being compiled,
2. the line number of the line containing the error (in parentheses),
3. a message number, and
4. text explaining the nature of the error.

In the above example, the first error occurred on line 6 of the file `ERR.C`. Error number 1034 (with the appropriate substitutions) was diagnosed. The second error occurred on line 9 of the file `ERR.C`. Error number 1011 (with the appropriate substitutions) was diagnosed.

The following sections contain a complete list of the messages. Run-time messages (messages displayed during execution) do not have message numbers associated with them.

B.1 Warning Level 1 Messages

W100 *Parameter %d contains inconsistent levels of indirection*

The function is expecting something like `char **` and it is being passed a `char *` for instance.

W101 *Non-portable pointer conversion*

This message is issued whenever you convert a non-zero constant to a pointer.

W102 *Type mismatch (warning)*

This message is issued for a function return value or an assignment where both types are pointers, but they are pointers to different kinds of objects.

W103 *Parameter count does not agree with previous definition (warning)*

You have either not enough parameters or too many parameters in a call to a function. If the function is supposed to have a variable number of parameters, then you can ignore this warning, or you can change the function declaration and prototypes to use the ",..." to indicate that the function indeed takes a variable number of parameters.

W104 *Inconsistent levels of indirection*

This occurs in an assignment or return statement when one of the operands has more levels of indirection than the other operand. For example, a `char **` is being assigned to a `char *`.

Solution: Correct the levels of indirection or use a `void *`.

W105 *Assignment found in boolean expression*

An assignment of a constant has been detected in a boolean expression. For example: "if(var = 0)". It is most likely that you want to use "==" for testing for equality.

W106 *Constant out of range - truncated*

This message is issued if a constant cannot be represented in 32 bits or if a constant is outside the range of valid values that can be assigned to a variable.

W107 *Missing return value for function '%s'*

A function has been declared with a function return type, but no ***return*** statement was found in the function. Either add a ***return*** statement or change the function return type to ***void.***

W108 *Duplicate typedef already defined*

A duplicate typedef is not allowed in ANSI C. This warning is issued when compiling with extensions enabled. You should delete the duplicate typedef definition.

W109 *not used*

unused message

W110 *'fortran' pragma not defined*

You have used the ***fortran*** keyword in your program, but have not defined a #pragma for ***fortran.***

W111 *Meaningless use of an expression*

The line contains an expression that does nothing useful. In the example "i = (1,5);", the expression "1," is meaningless.

W112 *Pointer truncated*

A far pointer is being passed to a function that is expecting a near pointer, or a far pointer is being assigned to a near pointer.

W113 *Pointer type mismatch*

You have two pointers that either point to different objects, or the pointers are of different size, or they have different modifiers.

W114 *Missing semicolon*

You are missing the semicolon ";" on the field definition just before the right curly brace "}".

W115 *&array may not produce intended result*

The type of the expression "&array" is different from the type of the expression "array". Suppose we have the declaration `char buffer[80]` Then the expression `(&buffer + 3)` will be evaluated as `(buffer + 3 * sizeof(buffer))` which is `(buffer + 3 * 80)` and not `(buffer + 3 * 1)` which is what most people expect to happen. The address of operator "&" is not required for getting the address of an array.

W116 *Attempt to return address of auto variable*

This warning usually indicates a serious programming error. When a function exits, the storage allocated on the stack for auto variables is released. This storage will be overwritten by further function calls and/or hardware interrupt service routines. Therefore, the data pointed to by the return value may be destroyed before your program has a chance to reference it or make a copy of it.

W117 *'##' tokens did not generate a single token (rest discarded)*

When two tokens are pasted together using ##, they must form a string that can be parsed as a single token.

W118 *Label '%s' has been defined but not referenced*

You have defined a label that is not referenced in a ***goto*** statement. It is possible that you are missing the ***case*** keyword when using an enumerated type name as a case in a ***switch*** statement. If not, then the label can be deleted.

W119 *Address of static function '%s' has been taken*

This warning may indicate a potential problem when the program is overlayed.

W120 *lvalue cast is not standard C*

A cast operation does not yield an lvalue in ANSI standard C. However, to provide compatibility with code written prior to the availability of ANSI standard C compilers, if an expression was an lvalue prior to the cast operation, and the cast operation does not cause any conversions, the compiler treats the result as an lvalue and issues this warning.

W121 *Text following pre-processor directives is not standard C*

Arbitrary text is not allowed following a pre-processor directive. Only comments are allowed following a pre-processor directive.

W122 *Literal string too long for array - truncated*

The supplied literal string contains more characters than the specified dimension of the array. Either shorten the literal string, or increase the dimension of the array to hold all of the characters from the literal string.

W123 *'//' style comment continues on next line*

The compiler has detected a line continuation during the processing of a C++ style comment ("//"). The warning can be removed by switching to a C style comment ("/**/"). If you require the comment to be terminated at the end of the line, make sure that the backslash character is not the last character in the line.

Example:

```
#define XX 23 // comment start \
comment \
end

int x = XX; // comment start ...\
comment end
```

W124 *Comparison result always %d*

The line contains a comparison that is alway true (1) or false (0). For example comparing an unsigned expression to see if it is >= 0 or < 0 is redundant. Check to see if the expression should be signed instead of unsigned.

W125 *Nested include depth of %d exceeded*

The number of nested include files has reached a preset limit, check for recursive include statements.

W126 *Constant must be zero for pointer compare*

A pointer is being compared using == or != to a non-zero constant.

W127 *trigraph found in string*

Trigraph expansion occurs inside a string literal. This warning can be disabled via the command line or ***#pragma warning*** directive.

Example:

```
// string expands to "(?]?????"!
char *e = "(???)???-????";
// possible work-arounds
char *f = "(" "???" ")" "???" "-" "????";
char *g = "(\?\?\?)\?\?\?-\?\?\?\?";
```

W128 *%d padding byte(s) added*

The compiler has added slack bytes to align a member to the correct offset.

W129 *#endif matches #if in different source file '%s'*

This warning may indicate a ***#endif*** nesting problem since the traditional usage of ***#if*** directives is confined to the same source file. This warning may often come before an error and it is hoped will provide information to solve a preprocessing directive problem.

B.2 Warning Level 2 Messages

W200 *'%s' has been referenced but never assigned a value*

You have used the variable in an expression without previously assigning a value to that variable.

W201 *Unreachable code*

The statement will never be executed, because there is no path through the program that causes control to reach this statement.

W202 *Symbol '%s' has been defined, but not referenced*

There are no references to the declared variable. The declaration for the variable can be deleted.

In some cases, there may be a valid reason for retaining the variable. You can prevent the message from being issued through use of *#pragma off(unreferenced).*

W203 *Preprocessing symbol '%s' has not been declared*

The symbol has been used in a preprocessor expression. The compiler assumes the symbol has a value of 0 and continues. A `#define` may be required for the symbol, or you may have forgotten to include the file which contains a `#define` for the symbol.

B.3 Warning Level 3 Messages

W300 *Nested comment found in comment started on line %u*

While scanning a comment for its end, the compiler detected `/*` for the start of another comment. Nested comments are not allowed in ANSI C. You may be missing the `*/` for the previous comment.

W301 *No prototype found for '%s'*

A reference for a function appears in your program, but you do not have a prototype for that function defined.

W302 *Expression is only useful for its side effects*

You have an expression that would have generated the warning "Meaningless use of an expression", except that it also contains a side-effect, such as ++, ––, or a function call.

W303 *Parameter '%s' has been defined, but not referenced*

There are no references to the declared parameter. The declaration for the parameter can be deleted. Since it is a parameter to a function, all calls to the function must also have the value for that parameter deleted.

In some cases, there may be a valid reason for retaining the parameter. You can prevent the message from being issued through use of *#pragma off(unreferenced).*

This warning is initially disabled. It must be specifically enabled with *#pragma enable_message(303).* It can be disabled later by using *#pragma disable_message(303).*

B.4 Error Messages

E1000 *BREAK must appear in while, do, for or switch statement*

A ***break*** statement has been found in an illegal place in the program. You may be missing an opening brace `{` for a ***while, do, for*** or ***switch*** statement.

E1001 *CASE must appear in switch statement*

A ***case*** label has been found that is not inside a ***switch*** statement.

E1002 *CONTINUE must appear in while, do or for statement*

The ***continue*** statement must be inside a ***while, do*** or ***for*** statement. You may have too many `}` between the ***while, do*** or ***for*** statement and the ***continue*** statement.

E1003 *DEFAULT must appear in switch statement*

A ***default*** label has been found that is not inside a ***switch*** statement. You may have too many `}` between the start of the ***switch*** and the ***default*** label.

E1004 *Misplaced '}' or missing earlier '{'*

An extra `}` has been found which cannot be matched up with an earlier `{`.

E1005 *Misplaced #elif directive*

The `#elif` directive must be inside an `#if` preprocessing group and before the `#else` directive if present.

E1006 *Misplaced #else directive*

The `#else` directive must be inside an `#if` preprocessing group and follow all `#elif` directives if present.

E1007 *Misplaced #endif directive*

A preprocessing directive has been found without a matching `#if` directive. You either have an extra or you are missing an `#if` directive earlier in the file.

E1008 *Only 1 DEFAULT per switch allowed*

You cannot have more than one ***default*** label in a ***switch*** statement.

E1009 *Expecting '%s' but found '%s'*

A syntax error has been detected. The tokens displayed in the message should help you to determine the problem.

E1010 *Type mismatch*

For pointer subtraction, both pointers must point to the same type. For other operators, both expressions must be assignment compatible.

E1011 *Symbol '%s' has not been declared*

The compiler has found a symbol which has not been previously declared. The symbol may be spelled differently than the declaration, or you may need to `#include` a header file that contains the declaration.

E1012 *Expression is not a function*

The compiler has found an expression that looks like a function call, but it is not defined as a function.

E1013 *Constant variable cannot be modified*

An expression or statement has been found which modifies a variable which has been declared with the ***const*** keyword.

E1014 *Left operand must be an 'lvalue'*

The operand on the left side of an "=" sign must be a variable or memory location which can have a value assigned to it.

E1015 *'%s' is already defined as a variable*

You are trying to declare a function with the same name as a previously declared variable.

E1016 *Expecting identifier*

The token following "->" and "." operators must be the name of an identifier which appears in the struct or union identified by the operand preceding the "->" and "." operators.

E1017 *Label '%s' already defined*

All labels within a function must be unique.

E1018 *Label '%s' not defined in function*

A ***goto*** statement has referenced a label that is not defined in the function. Add the necessary label or check the spelling of the label(s) in the function.

E1019 *Tag '%s' already defined*

All ***struct, union*** and ***enum*** tag names must be unique.

E1020 *Dimension cannot be 0 or negative*

The dimension of an array must be positive and non-zero.

E1021 *Dimensions of multi-dimension array must be specified*

All dimensions of a multiple dimension array must be specified. The only exception is the first dimension which can declared as "[]".

E1022 *Missing or misspelled data type near '%s'*

The compiler has found an identifier that is not a predefined type or the name of a "typedef". Check the identifier for a spelling mistake.

E1023 *Storage class of parameter must be register or unspecified*

The only storage class allowed for a parameter declaration is ***register.***

E1024 *Declared symbol '%s' is not in parameter list*

Make sure that all the identifiers in the parameter list match those provided in the declarations between the start of the function and the opening brace "{".

E1025 *Parameter '%s' already declared*

A declaration for the specified parameter has already been processed.

E1026 *Invalid declarator*

A syntax error has occurred while parsing a declaration.

E1027 *Invalid storage class for function*

If a storage class is given for a function, it must be ***static*** or ***extern.***

E1028 *Variable '%s' cannot be void*

You cannot declare a ***void*** variable.

E1029 *Expression must be 'pointer to ...'*

An attempt has been made to de-reference (*) a variable or expression which is not declared to be a pointer.

E1030 *Cannot take the address of an rvalue*

You can only take the address of a variable or memory location.

E1031 *Name '%s' not found in struct/union %s*

The specified identifier is not one of the fields declared in the ***struct*** or ***union.*** Check that the field name is spelled correctly, or that you are pointing to the correct ***struct*** or ***union.***

E1032 *Expression for '.' must be a 'structure' or 'union'*

The compiler has encountered the pattern "expression" "." "field_name" where the expression is not a ***struct*** or ***union*** type.

E1033 *Expression for '->' must be 'pointer to struct or union'*

The compiler has encountered the pattern "expression" "->" "field_name" where the expression is not a pointer to ***struct*** or ***union*** type.

E1034 *Symbol '%s' already defined*

The specified symbol has already been defined.

E1035 *static function '%s' has not been defined*

A prototype has been found for a ***static*** function, but a definition for the ***static*** function has not been found in the file.

E1036 *Right operand of '%s' is a pointer*

The right operand of "+=" and "−=" cannot be a pointer. The right operand of "−" cannot be a pointer unless the left operand is also a pointer.

E1037 *Type cast must be a scalar type*

You cannot type cast an expression to be a ***struct, union,*** array or function.

E1038 *Expecting label for goto statement*

The ***goto*** statement requires the name of a label.

E1039 *Duplicate case value '%s' found*

Every case value in a ***switch*** statement must be unique.

E1040 *Field width too large*

The maximum field width allowed is 16 bits.

E1041 *Field width of 0 with symbol not allowed*

A bit field must be at least one bit in size.

E1042 *Field width must be positive*

You cannot have a negative field width.

E1043 *Invalid type specified for bit field*

The types allowed for bit fields are ***signed*** or ***unsigned*** varieties of ***char, short*** and ***int.***

E1044 *Variable '%s' has incomplete type*

A full definition of a ***struct*** or ***union*** has not been given.

E1045 *Subscript on non-array*

One of the operands of "[]" must be an array.

E1046 *Incomplete comment*

The compiler did not find `*/` to mark the end of a comment.

E1047 *Argument for # must be a macro parm*

The argument for the stringize operator "#" must be a macro parameter.

E1048 *Unknown preprocessing directive '#%s'*

An unrecognized preprocessing directive has been encountered. Check for correct spelling.

E1049 *Invalid #include directive*

A syntax error has been encountered in a `#include` directive.

E1050 *Not enough parameters given for macro '%s'*

You have not supplied enough parameters to the specified macro.

E1051 *Not expecting a return value for function '%s'*

The specified function is declared as a ***void*** function. Delete the ***return*** statement, or change the type of the function.

E1052 *Expression has void type*

You tried to use the value of a ***void*** expression inside another expression.

E1053 *Cannot take the address of a bit field*

The smallest addressable unit is a byte. You cannot take the address of a bit field.

E1054 *Expression must be constant*

The compiler expects a constant expression. This message can occur during static initialization if you are trying to initialize a non-pointer type with an address expression.

E1055 *Unable to open '%s'*

The file specified in an `#include` directive could not be located. Make sure that the file name is spelled correctly, or that the appropriate path for the file is included in the list of paths specified in the `INCLUDE` environment variable or the "i=" option on the command line.

E1056 *Too many parameters given for macro '%s'*

You have supplied too many parameters for the specified macro.

E1057 *Modifiers disagree with previous definition of '%s'*

You have more than one definition or prototype for the variable or function which have different type modifiers.

E1058 *Cannot use typedef '%s' as a variable*

The name of a typedef has been found when an operand or operator is expected. If you are trying to use a type cast, make sure there are parentheses around the type, otherwise check for a spelling mistake.

E1059 *Invalid storage class for non-local variable*

A variable with module scope cannot be defined with the storage class of ***auto*** or ***register.***

E1060 *Invalid type*

An invalid combination of the following keywords has been specified in a type declaration: ***const, volatile, signed, unsigned, char, int, short, long, float*** and ***double.***

E1061 *Expecting data or function declaration, but found '%s'*

The compiler is expecting the start of a data or function declaration. If you are only part way through a function, then you have too many closing braces "}".

E1062 *Inconsistent return type for function '%s'*

Two prototypes for the same function disagree.

E1063 *Missing operand*

An operand is required in the expression being parsed.

E1064 *Out of memory*

The compiler has run out of memory to store information about the file being compiled. Try reducing the number of data declarations and or the size of the file being compiled. Do not `#include` header files that are not required.

For the 16-bit WATCOM C compiler, the "/d2" switch causes the compiler to use more memory. Try compiling with the "/d1" switch instead.

E1065 *Invalid character constant*

This message is issued for an improperly formed character constant.

E1066 *Cannot perform operation with pointer to void*

You cannot use a "pointer to void" with the operators +, –, ++, ––, += and –=.

E1067 *Cannot take address of variable with storage class 'register'*

If you want to take the address of a local variable, change the storage class from ***register*** to ***auto.***

E1068 *Variable '%s' already initialized*

The specified variable has already been statically initialized.

E1069 *Ending \" missing for string literal*

The compiler did not find a second double quote to end the string literal.

E1070 *Data for aggregate type must be enclosed in curly braces*

When an array, struct or union is statically initialized, the data must be enclosed in curly braces {}.

E1071 *Type of parameter %d does not agree with previous definition*

The type of the specified parameter is incompatible with the prototype for that function. The following example illustrates a problem that can arise when the sequence of declarations is in the wrong order.

Example:

```
/* Uncommenting the following line will
   eliminate the error */
/* struct foo; */

void fn1( struct foo * );

struct foo {
    int     a,b;
};

void fn1( struct foo *bar )
{
    fn2( bar );
}
```

The problem can be corrected by reordering the sequence in which items are declared (by moving the description of the structure `foo` ahead of its first reference or by adding the indicated statement). This will assure that the first instance of structure `foo` is defined at the proper outer scope.

E1072 *Storage class disagrees with previous definition of '%s'*

The previous definition of the specified variable has a storage class of ***static.*** The current definition must have a storage class of ***static*** or ***extern.***

E1073 *Invalid option '%s'*

The specified option is not recognized by the compiler.

E1074 *Invalid optimization option '%s'*

The specified option is an unrecognized optimization option.

E1075 *Invalid memory model '%s'*

Memory model option must be one of "ms", "mm", "mc", "ml", "mh" or "mf" which selects the Small, Medium, Compact, Large, Huge or Flat memory model.

E1076 *Missing semicolon at end of declaration*

You are missing a semicolon ";" on the declaration just before the left curly brace "{".

E1077 *Missing '}'*

The compiler detected end of file before finding a right curly brace "}" to end the current function.

E1078 *Invalid type for switch expression*

The type of a switch expression must be integral.

E1079 *Expression must be integral*

An integral expression is required.

E1080 *Expression must be arithmetic*

Both operands of the "*", "/" and "%" operators must be arithmetic. The operand of the unary minus must also be arithmetic.

E1081 *Expression must be scalar type*

A scalar expression is required.

E1082 *Statement required after label*

The C language definition requires a statement following a label. You can use a null statement which consists of just a semicolon (";").

E1083 *Statement required after 'do'*

A statement is required between the ***do*** and ***while*** keywords.

E1084 *Statement required after 'case'*

The C language definition requires a statement following a ***case*** label. You can use a null statement which consists of just a semicolon (";").

E1085 *Statement required after 'default'*

The C language definition requires a statement following a ***default*** label. You can use a null statement which consists of just a semicolon (";").

E1086 *Expression too complicated, split it up and try again*

The expression contains too many levels of nested parentheses. Divide the expression up into two or more sub-expressions.

E1087 *Missing matching #endif directive*

You are missing a to terminate a `#if, #ifdef` or `#ifndef` preprocessing directive.

E1088 *Invalid macro definition, missing)*

The right parenthesis ")" is required for a function-like macro definition.

E1089 *Missing) for expansion of '%s' macro*

The compiler encountered end-of-file while collecting up the argument for a function-like macro. A right parenthesis ")" is required to mark the end of the argument(s) for a function-like macro.

E1090 *Invalid conversion*

A ***struct*** or ***union*** cannot be converted to anything. A ***float*** or ***double*** cannot be converted to a pointer and a pointer cannot be converted to a ***float*** or ***double.***

E1091 *%s*

This is a user message generated with the `#error` preprocessing directive.

E1092 *Cannot define an array of functions*

You can have an array of pointers to functions, but not an array of functions.

E1093 *Function cannot return an array*

A function cannot return an array. You can return a pointer to an array.

E1094 *Function cannot return a function*

You cannot return a function. You can return a pointer to a function.

E1095 *Cannot take address of local variable in static initialization*

You cannot take the address of an ***auto*** variable at compile time.

E1096 *Inconsistent use of return statements*

The compiler has found a ***return*** statement which returns a value and a ***return*** statement that does not return a value both in the same function. The ***return*** statement which does not return a value needs to have a value specified to be consistent with the other ***return*** statement in the function.

E1097 *Missing ? or misplaced :*

The compiler has detected a syntax error related to the "?" and ":" operators. You may need parenthesis around the expressions involved so that it can be parsed correctly.

E1098 *Maximum struct or union size is 64K*

The size of a ***struct*** or ***union*** is limited to 64K so that the compiler can represent the offset of a member in a 16-bit register.

E1099 *Statement must be inside function. Probable cause: missing {*

The compiler has detected a statement such as ***for, while, switch,*** etc., which must be inside a function. You either have too many closing braces "}" or you are missing an opening brace "{" earlier in the function.

E1100 *Definition of macro '%s' not identical to previous definition*

If a macro is defined more than once, the definitions must be identical. If you want to redefine a macro to have a different definition, you must `#undef` it before you can define it with a new definition.

E1101 *Cannot #undef '%s'*

The special macros `__LINE__, __FILE__, __DATE__, __TIME__,` and `__STDC__,` and the identifier "defined", cannot be deleted by the `#undef` directive.

E1102 *Cannot #define the name 'defined'*

You cannot define a macro called `defined.`

E1103 *## must not be at start or end of replacement tokens*

There must be a token on each side of the "##" (token pasting) operator.

E1104 *Type cast not allowed in #if or #elif expression*

A type cast is not allowed in a preprocessor expression.

E1105 *'sizeof' not allowed in #if or #elif expression*

The ***sizeof*** operator is not allowed in a preprocessor expression.

E1106 *Cannot compare a struct or union*

A ***struct*** or ***union*** cannot be compared with "==" or "`!=`". You must compare each member of a ***struct*** or ***union*** to determine equality or inequality. If the ***struct*** or ***union*** is packed (has no holes in it for alignment purposes) then you can compare two structs using `memcmp.`

E1107 *Enumerator list cannot be empty*

You must have at least one identifier in an ***enum*** list.

E1108 *Invalid floating-point constant*

The exponent part of the floating-point constant is not formed correctly.

E1109 *Cannot take sizeof a bit field*

The smallest object that you can ask for the size of is a char.

E1110 *Cannot initialize variable with storage class of extern*

A storage class of ***extern*** is used to associate the variable with its actual definition somewhere else in the program.

E1111 *Invalid storage class for parameter*

The only storage class allowed for a parameter is ***register.***

E1112 *Initializer list cannot be empty*

An initializer list must have at least one item specified.

E1113 *Expression has incomplete type*

An attempt has been made to access a struct or union whose definition is not known, or an array whose dimensions are not known.

E1114 *Struct or union cannot contain itself*

You cannot have a ***struct*** or ***union*** contain itself. You can have a pointer in the ***struct*** which points to an instance of itself. Check for a missing "*" in the declaration.

E1115 *Incomplete enum declaration*

The enumeration tag has not been previously defined.

E1116 *An id list not allowed except for function definition*

A function prototype must contain type information.

E1117 *Must use 'va_start' macro inside function with variable parameters*

The `va_start` macro is used to setup access to the parameters in a function that takes a variable number of parameters. A function is defined with a variable number of parameters by declaring the last parameter in the function as "...".

E1118 ****FATAL*** %s*

A fatal error has been detected during code generation time. The type of error is displayed in the message.

E1119 *Internal compiler error %d*

A bug has been encountered in the WATCOM C compiler. Please report the specified internal compiler error number and any other helpful details about the program being compiled to WATCOM so that we can fix the problem.

E1120 *Parameter number %d - invalid register in #pragma*

The designated registers cannot hold the value for the parameter.

E1121 *Procedure '%s' has invalid return register in #pragma*

The size of the return register does not match the size of the result returned by the function.

E1122 *Illegal register modified by '%s' #pragma*

For the 16-bit WATCOM C compiler: The BP, CS, DS, and SS registers cannot be modified in small data models. The BP, CS, and SS registers cannot be modified in large data models.

For the 32-bit WATCOM C compiler: The EBP, CS, DS, ES, and SS registers cannot be modified in flat memory models. The EBP, CS, DS, and SS registers cannot be modified in small data models. The EBP, CS, and SS registers cannot be modified in large data models.

E1123 *File must contain at least one external definition*

Every file must contain at least one global object, (either a data variable or a function). This message is only issued in strict ANSI mode (-za).

E1124 *Out of macro space*

The compiler ran out of memory for storing macro definitions.

E1125 *Keyboard interrupt detected*

The compile has been aborted with Ctrl/C or Ctrl/Break.

E1126 *Array, struct or union cannot be placed in a register*

Only scalar objects can be specified with the ***register*** class.

E1127 *Type required in parameter list*

If the first parameter in a function definition or prototype is defined with a type, then all of the parameters must have a type specified.

E1128 *Enum constant too large*

All of the constants must fit in either an ***int*** or ***unsigned.***

E1129 *Type does not agree with previous definition of '%s'*

You have more than one definition of a variable or function that do not agree.

E1130 *Duplicate name '%s' not allowed in struct or union*

All the field names in a ***struct*** or ***union*** must be unique.

E1131 *Duplicate macro parameter '%s'*

The parameters specified in a macro definition must be unique.

E1132 *Unable to open work file: error code = %d*

The compiler tries to open a new work file by the name "__wrkN__.tmp" where N is the digit 0 to 9. This message will be issued if all of those files already exist.

E1133 *Write error on work file: error code = %d*

An error was encountered trying to write information to the work file. The disk could be full.

E1134 *Read error on work file: error code = %d*

An error was encountered trying to read information from the work file.

E1135 *Seek error on work file: error code = %d*

An error was encountered trying to seek to a position in the work file.

E1136 *Token too long - truncated*

The token must be less than 510 bytes in length.

E1137 *Out of enum space*

The compiler has run out of space allocated to store information on all of the ***enum*** constants defined in your program.

E1138 *Filename required on command line*

The name of a file to be compiled must be specified on the command line.

E1139 *Command line contains more than one file to compile*

You have more than one file name specified on the command line to be compiled. The compiler can only compile one file at a time. You can use the Watcom Compile and Link utility to compile multiple files with a single command.

E1140 *_leave must appear in a _try statement*

The ***_leave*** keyword must be inside a ***_try*** statement. The ***_leave*** keyword causes the program to jump to the start of the ***_finally*** block.

E1141 *Expecting end of line but found '%s'*

A syntax error has been detected. The token displayed in the message should help you determine the problem.

E1142 *Too many bytes specified in #pragma*

There is an internal limit on the number of bytes for in-line code that can be specified with a pragma. Try splitting the function into two or more smaller functions.

E1143 *Cannot resolve linkage conventions for routine '%s' #pragma*

The compiler cannot generate correct code for the specified routine because of register conflicts. Change the registers used by the parameters of the pragma.

E1144 *Symbol '%s' in pragma must be global*

The in-line code for a pragma can only reference a global variable or function. You can only reference a parameter or local variable by passing it as a parameter to the in-line code pragma.

E1145 *Internal compiler limit exceeded, break module into smaller pieces*

The compiler can handle 65535 quadruples, 65535 leaves, and 65535 symbol table entries and literal strings. If you exceed one of these limits, the program must be broken into smaller pieces until it is capable of being processed by the compiler.

E1146 *Invalid initializer for integer data type*

Integer data types (int and long) can be initialized with numeric expressions or address expressions that are the same size as the integer data type being initialized.

E1147 *Too many errors: compilation aborted*

The compiler stops compiling when the number of errors generated exceeds the error limit. The error limit can be set with the "-e" option. The default error limit is 20.

E1148 *Expecting identifier but found '%s'*

A syntax error has been detected. The token displayed in the message should help you determine the problem.

E1149 *Expecting constant but found '%s'*

The #line directive must be followed by a constant indicating the desired line number.

E1150 *Expecting \"filename\" but found '%s'*

The second argument of the #line directive must be a filename enclosed in quotes.

E1151 *Parameter count does not agree with previous definition*

You have either not enough parameters or too many parameters in a call to a function. If the function is supposed to have a variable number of parameters, then you are missing the ", ..." in the function prototype.

E1152 *Segment name required*

A segment name must be supplied in the form of a literal string to the __segname() directive.

E1153 *Invalid __based declaration*

The compiler could not recognize one of the allowable forms of __based declarations. See the *WATCOM C Language Reference* for description of all the allowable forms of __based declarations.

E1154 *Variable for __based declaration must be of type __segment or pointer*

A based pointer declaration must be based on a simple variable of type __segment or pointer.

E1155 *Duplicate external symbol %s*

Duplicate external symbols will exist when the specified symbol name is truncated to 8 characters.

E1156 *Assembler error: '%s'*

An error has been detected by the in-line assembler. The message indicates the error detected.

E1157 *Variable must be 'huge'*

A variable or an array that requires more than 64K of storage in the 16-bit compiler must be declared as ***huge.***

E1158 *Too many parm sets*

Too many parameter register sets have been specified in the pragma.

E1159 *I/O error reading '%s': %s*

An I/O error has been detected by the compiler while reading the source file. The system dependent reason is also displayed in the message.

E1160 *Attempt to access far memory with all segment registers disabled in '%s'*

The compiler does not have any segment registers available to access the desired far memory location.

E1161 *No identifier provided for /D option*

The command line option /D must be followed by the name of the macro to be defined.

E1162 *Invalid register pegged to a segment in '%s'*

The register specified in a #pragma data_seg, or a ***__segname*** expression must be a valid segment register.

E1163 *Invalid octal constant*

An octal constant cannot contain the digits 8 or 9.

E1164 *Invalid hexadecimal constant*

The token sequence "0x" must be followed by a hexadecimal character (0-9, a-f, or A-F).

E1165 *Unexpected ')'. Probable cause: missing '('*

A closing parenthesis was found in an expression without a corresponding opening parenthesis.

E1166 *Symbol '%s' is unreachable from #pragma*

The in-line assembler found a jump instruction to a label that is too far away.

E1167 *Division or remainder by zero in a constant expression*

The compiler found a constant expression containing a division or remainder by zero.

E1168 *Cannot end string literal with backslash*

The argument to a macro that uses the stringize operator '#' on that argument must not end in a backslash character.

Example:

```
#define str(x) #x
str(@#\)
```

E1169 *Invalid __declspec declaration*

The only valid __declspec declarations are "__declspec(thread)", "__declspec(dllexport)", and "__declspec(dllimport)".

E1170 *Too many storage class specifiers*

You can only specify one storage class specifier in a declaration.

E1171 *Expecting '%s' but found end of file*

A syntax error has been detected. The compiler is still expecting more input when it reached the end of the source program.

E1172 *Expecting struct/union tag but found '%s'*

The compiler expected to find an identifier following the ***struct*** or ***union*** keyword.

E1173 *Operand of __builtin_isfloat() must be a type*

The __builtin_isfloat() function is used by the ***va_arg*** macro to determine if a type is a floating-point type.

E1174 *Invalid constant*

The token sequence does not represent a valid numeric constant.

E1175 *Too many initializers*

There are more initializers than objects to initialize. For example int X[2] = { 0, 1, 2 }; The variable "X" requires two initializers not three.

E1176 *Parameter %d, pointer type mismatch*

You have two pointers that either point to different objects, or the pointers are of different size, or they have different modifiers.

E1177 *Modifier repeated in declaration*

You have repeated the use of a modifier like "const" (an error) or "far" (a warning) in a declaration.

E1178 *Type qualifier mismatch*

You have two pointers that have different "const" or "volatile" qualifiers.

E1179 *Parameter %d, type qualifier mismatch*

You have two pointers that have different const or "volatile" qualifiers.

E1180 *Sign specifier mismatch*

You have two pointers that point to types that have different sign specifiers.

E1181 *Parameter %d, sign specifier mismatch*

You have two pointers that point to types that have different sign specifiers.

E1182 *Missing \\ for string literal*

You need a '\' to continue a string literal across a line.

B.5 Informational Messages

I2000 *Not enough memory to fully optimize procedure '%s'*

The compiler did not have enough memory to fully optimize the specified procedure. The code generated will still be correct and execute properly. This message is purely informational.

I2001 *Not enough memory to maintain full peephole*

Certain optimizations benefit from being able to store the entire module in memory during optimization. All functions will be individually optimized but the optimizer will not be able to share code between functions if this message appears. The code generated will still be correct and execute properly. This message is purely informational. It is only printed if the warning level is greater than or equal to 4.

The main reason for this message is for those people who are concerned about reproducing the exact same object code when the same source file is compiled on a different machine. You may not be able to reproduce the exact same object code from one compile to the next unless the available memory is exactly the same.

B.6 Pre-compiled Header Messages

H3000 *Error reading PCH file*

The pre-compiled header file does not follow the correct format.

H3001 *PCH file header is out of date*

The pre-compiled header file is out of date with the compiler. The current version of the compiler is expecting a different format.

H3002 *Compile options differ with PCH file*

The command line options are not the same as used when making the pre-compiled header file. This can effect the values of the pre-compiled information.

H3003 *Current working directory differs with PCH file*

The pre-compiled header file was compiled in a different directory.

H3004 *Include file '%s' has been modified since PCH file was made*

The include files have been modified since the pre-compiled header file was made.

H3005 *PCH file was made from a different include file*

The pre-compiled header file was made using a different include file.

H3006 *Include path differs with PCH file*

The include paths have changed.

H3007 *Preprocessor macro definition differs with PCH file*

The definition of a preprocessor macro has changed.

H3008 *PCH cannot have data or code definitions.*

The include files used to build the pre-compiled header contain function or data definitions. This is not currently supported.

B.7 Miscellaneous Messages and Phrases

M4000 *Code size*

String used in message construction.

M4001 *Error!*

String used in message construction.

M4002 *Warning!*

String used in message construction.

M4003 *Note!*

String used in message construction.

M4004 *(Press return to continue)*

String used in message construction.

C. Watcom C++ Diagnostic Messages

The following is a list of all warning and error messages produced by the Watcom C++ compilers. Diagnostic messages are issued during compilation and execution.

The messages listed in the following sections contain references to `%N`, `%S`, `%T`, `%s`, `%d` and `%u`. They represent strings that are substituted by the Watcom C++ compilers to make the error message more exact. `%d` and `%u` represent a string of digits; `%N`, `%S`, `%T` and `%s` a string, usually a symbolic name.

Consider the following program, named `ERR.CPP`, which contains errors.

Example:

```
#include <stdio.h>

void main()
  {
    int i;
    float i;

    i = 383;
    x = 13143.0;
    printf( "Integer value is %d\n", i );
    printf( "Floating-point value is %f\n", x );
  }
```

If we compile the above program, the following messages will appear on the screen.

```
File: err.cpp
(6,12): Error! E042: symbol 'i' already defined
  'i' declared at: (5,9)
(9,5): Error! E029: symbol 'x' has not been declared
err.cpp: 12 lines, included 174, no warnings, 2 errors
```

The diagnostic messages consist of the following information:

1. the name of the file being compiled,
2. the line number and column of the line containing the error (in parentheses),
3. a message number, and

4. text explaining the nature of the error.

In the above example, the first error occurred on line 6 of the file `ERR.CPP`. Error number 042 (with the appropriate substitutions) was diagnosed. The second error occurred on line 9 of the file `ERR.CPP`. Error number 029 (with the appropriate substitutions) was diagnosed.

The following sections contain a complete list of the messages. Run-time messages (messages displayed during execution) do not have message numbers associated with them.

A number of messages contain a reference to the ARM. This is the "Annotated C++ Reference Manual" written by Margaret A. Ellis and Bjarne Stroustrup and published by Addison-Wesley (ISBN 0-201-51459-1).

C.1 Diagnostic Messages

000 *internal compiler error*

If this message appears, please report the problem directly to Watcom.

001 *assignment of constant found in boolean expression*

An assignment of a constant has been detected in a boolean expression. For example: "if(var = 0)". It is most likely that you want to use "==" for testing for equality.

002 *constant out of range; truncated*

This message is issued if a constant cannot be represented in 32 bits or if a constant is outside the range of valid values that can be assigned to a variable.

Example:

```
int a = 12345678901234567890;
```

003 *missing return value*

A function has been declared with a function return type, but no ***return*** statement was found in the function. Either add a ***return*** statement or change the function return type to ***void.***

Example:

```
int foo( int a )
{
    int b = a + a;
}
```

The message will be issued at the end of the function.

004 *base class '%T' does not have a virtual destructor*

A virtual destructor has been declared in a class with base classes. The compiler has detected that a base class does not have a virtual destructor, so a ***delete*** of a pointer cast to the base class will not function properly in all circumstances.

Example:

```
struct Base {
    ~Base();
};
struct Derived : Base {
    virtual ~Derived();
};
```

It is usually considered good programming practice to declare virtual destructors in all classes used as base classes of classes having virtual destructors.

005 *pointer or reference truncated*

The expression contains a transfer of a pointer value to another pointer value of smaller size. This can be caused by ***__near*** or ***__far*** qualifiers (i.e., assigning a ***far*** pointer to a ***near*** pointer). Function pointers can also have a different size than data pointers in certain memory models. This message indicates that some information is being lost so check the code carefully.

Example:

```
extern int __far *foo();
int __far *p_far = foo();
int __near *p_near = p_far; // truncated
```

006 *syntax error; probable cause: missing ';'*

The compiler has found a complete expression (or declaration) during parsing but could not continue. The compiler has detected that it could have continued if a semicolon was present so there may be a semicolon missing.

Example:

```
enum S {
}       // missing ';'

class X {
};
```

007 *"&array" may not produce intended result*

The type of the expression "&array" is different from the type of the expression "array". Suppose we have the declaration `char buffer[80].` Then the expression `(&buffer + 3)` will be evaluated as `(buffer + 3 * sizeof(buffer))` which is `(buffer + 3 * 80)` and not `(buffer + 3 * 1)` which is what most people expect to happen. The address of operator "&" is not required for getting the address of an array.

008 *returning address of function argument or of auto or register variable*

This warning usually indicates a serious programming error. When a function exits, the storage allocated on the stack for auto variables is released. This storage will be overwritten by further function calls and/or hardware interrupt service routines. Therefore, the data pointed to by the return value may be destroyed before your program has a chance to reference it or make a copy of it.

Example:

```
int *foo()
{
    int k = 123;
    return &k;      // k is automatic variable
}
```

009 *option requires a file name*

The specified option is not recognized by the compiler since there was no file name after it (i.e., "-fo=my.obj").

010 *asm directive ignored*

The asm directive (e.g., asm("mov r0,1");) is a non-portable construct. The Watcom C++ compiler treats all asm directives like comments.

011 *all members are private*

This message warns the programmer that there will be no way to use the contents of the class because all accesses will be flagged as erroneous (i.e., accessing a private member).

Example:

```
class Private {
    int a;
    Private();
    ~Private();
    Private( const Private& );
};
```

012 *template argument cannot be type '%T'*

A template argument can be either a generic type (e.g., `template < class T >`), a pointer, or an integral type. These types are required for expressions that can be checked at compile time.

013 *unreachable code*

The statement will never be executed, because there is no path through the program that causes control to reach this statement.

Example:

```
void foo( int *p )
{
    *p = 4;
    return;
    *p = 6;
}
```

The statement following the ***return*** statement cannot be reached.

014 *no reference to symbol '%S'*

There are no references to the declared variable. The declaration for the variable can be deleted. If the variable is a parameter to a function, all calls to the function must also have the value for that parameter deleted.

In some cases, there may be a valid reason for retaining the variable. You can prevent the message from being issued through use of *#pragma off(unreferenced)*, or adding a statement that assigns the variable to itself.

015 *nested comment found in comment started on line %u*

While scanning a comment for its end, the compiler detected `/*` for the start of another comment. Nested comments are not allowed in ISO/ANSI C. You may be missing the `*/` for the previous comment.

016 *template argument list cannot be empty*

An empty template argument list would result in a template that could only define a single class or function.

017 *label '%s' has not been referenced by a goto*

The indicated label has not been referenced and, as such, is useless. This warning can be safely ignored.

Example:

```
int foo( int a, int b )
{
un_refed:
    return a + b;
}
```

018 *no reference to anonymous union member '%S'*

The declaration for the anonymous member can be safely deleted without any effect.

019 *'break' may only appear in a for, do, while, or switch statement*

A ***break*** statement has been found in an illegal place in the program. You may be missing an opening brace `{` for a ***while, do, for*** or ***switch*** statement.

Example:

```
int foo( int a, int b )
{
    break;      // illegal
    return a+b;
}
```

020 *'case' may only appear in a switch statement*

A ***case*** label has been found that is not inside a ***switch*** statement.

Example:

```
int foo( int a, int b )
{
    case 4:    // illegal
    return a+b;
}
```

021 *'continue' may only appear in a for, do, or while statement*

The ***continue*** statement must be inside a ***while, do*** or ***for*** statement. You may have too many `}` between the ***while, do*** or ***for*** statement and the ***continue*** statement.

Example:

```
int foo( int a, int b )
{
    continue;   // illegal
    return a+b;
}
```

022 *'default' may only appear in a switch statement*

A ***default*** label has been found that is not inside a ***switch*** statement. You may have too many `}` between the start of the ***switch*** and the ***default*** label.

Example:

```
int foo( int a, int b )
{
    default: // illegal
    return a+b;
}
```

023 *misplaced '}' or missing earlier '{'*

An extra `}` has been found which cannot be matched up with an earlier `{`.

024 *misplaced #elif directive*

The ***#elif*** directive must be inside an ***#if*** preprocessing group and before the ***#else*** directive if present.

Example:

```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

025 *misplaced #else directive*

The ***#else*** directive must be inside an ***#if*** preprocessing group and follow all ***#elif*** directives if present.

Example:

```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

026 *misplaced #endif directive*

A ***#endif*** preprocessing directive has been found without a matching ***#if*** directive. You either have an extra ***#endif*** or you are missing an ***#if*** directive earlier in the file.

Example:

```
int a;
#else
int c;
#elif IN_IF
int b;
#endif
```

The ***#else, #elif,*** and ***#endif*** statements are all illegal because there is no ***#if*** that corresponds to them.

027 *only one 'default' per switch statement is allowed*

You cannot have more than one ***default*** label in a ***switch*** statement.

Example:

```
int translate( int a )
{
    switch( a ) {
      case 1:
        a = 8;
        break;
      default:
        a = 9;
        break;
      default: // illegal
        a = 10;
        break;
    }
    return a;
}
```

028 *expecting '%s' but found '%s'*

A syntax error has been detected. The tokens displayed in the message should help you to determine the problem.

029 *symbol '%N' has not been declared*

The compiler has found a symbol which has not been previously declared. The symbol may be spelled differently than the declaration, or you may need to ***#include*** a header file that contains the declaration.

Example:

```
int a = b;  // b has not been declared
```

030 *left expression must be a function or a function pointer*

The compiler has found an expression that looks like a function call, but it is not defined as a function.

Example:

```
int a;
int b = a( 12 );
```

031 *operand must be an 'lvalue'*

The operand on the left side of an "=" sign must be a variable or memory location which can have a value assigned to it.

Example:

```
void foo( int a )
{
    ( a + 1 ) = 7;
    int b = ++ ( a + 6 );
}
```

Both statements within the function are erroneous, since "lvalues" are expected where the additions are shown.

032 *label '%s' already defined*

All labels within a function must be unique.

Example:
```
void bar( int *p )
{
label:
    *p = 0;
label:
    return;
}
```

The second label is illegal.

033 *label '%s' is not defined in function*

A ***goto*** statement has referenced a label that is not defined in the function. Add the necessary label or check the spelling of the label(s) in the function.

Example:
```
void bar( int *p )
{
    *p = 0;
    goto label;
}
```

The label referenced in the ***goto*** is not defined.

034 *dimension cannot be zero*

The dimension of an array must be non-zero.

Example:
```
int array[0];   // not allowed
```

035 *dimension cannot be negative*

The dimension of an array must be positive.

Example:

```
int array[-1];  // not allowed
```

036 *dimensions of multi-dimension array must be specified*

All dimensions of a multiple dimension array must be specified. The only exception is the first dimension which can declared as "[]".

Example:

```
int array[][];  // not allowed
```

037 *invalid storage class for function*

If a storage class is given for a function, it must be ***static*** or ***extern.***

Example:

```
auto void foo()
{
}
```

038 *expression must be 'pointer to ...'*

An attempt has been made to de-reference (*) a variable or expression which is not declared to be a pointer.

Example:

```
int a;
int b = *a;
```

039 *cannot take address of an 'rvalue'*

You can only take the address of a variable or memory location.

Example:

```
char c;
char *p1 = & & c;   // not allowed
char *p2 = & (c+1); // not allowed
```

040 *expression for '.' must be a class, struct or union*

The compiler has encountered the pattern "expression" "." "field_name" where the expression is not a ***class, struct*** or ***union*** type.

Example:

```
struct S
{
    int a;
};
int &fun();
int a = fun().a;
```

041 *expression for '->' must be pointer to class, struct or union*

The compiler has encountered the pattern "expression" "->" "field_name" where the expression is not a pointer to ***class, struct*** or ***union*** type.

Example:

```
struct S
{
    int a;
};
int *fun();
int a = fun()->a;
```

042 *symbol '%S' already defined*

The specified symbol has already been defined.

Example:

```
char a = 2;
char a = 2;     // not allowed
```

043 *static function '%S' has not been defined*

A prototype has been found for a ***static*** function, but a definition for the ***static*** function has not been found in the file.

Example:

```
static int fun( void );
int k = fun();
// fun not defined by end of program
```

044 *expecting label for goto statement*

The ***goto*** statement requires the name of a label.

Example:

```
int fun( void )
{
    goto;
}
```

045 *duplicate case value '%s' found*

Every case value in a ***switch*** statement must be unique.

Example:

```
int fun( int a )
{
    switch( a ) {
      case 1:
        return 7;
      case 2:
        return 9;
      case 1: // duplicate not allowed
        return 7;
    }
    return 79;
}
```

046 *bit-field width is too large*

The maximum field width allowed is 16 bits in the 16-bit compiler and 32 bits in the 32-bit compiler.

Example:

```
struct S
{
    unsigned bitfield :48;  // too wide
};
```

047 *width of a named bit-field must not be zero*

A bit field must be at least one bit in size.

Example:

```
struct S {
    int bitfield :10;
    int :0;   // ok, aligns to int
    int h :0; // err, field is named
};
```

048 *bit-field width must be positive*

You cannot have a negative field width.

Example:

```
struct S
{
    unsigned bitfield :-10; // cannot be negative
};
```

049 *bit-field base type must be an integral type*

The types allowed for bit fields are ***signed*** or ***unsigned*** varieties of ***char, short*** and ***int.***

Example:

```
struct S
{
    float bitfield : 10;    // must be integral
};
```

050 *subscript on non-array*

One of the operands of "[]" must be an array or a pointer.

Example:

```
int array[10];
int i1 = array[0];  // ok
int i2 = 0[array];  // same as above
int i3 = 0[1];      // illegal
```

051 *incomplete comment*

The compiler did not find `*/` to mark the end of a comment.

052 *argument for # must be a macro parm*

The argument for the stringize operator "#" must be a macro parameter.

053 *unknown preprocessing directive '#%s'*

An unrecognized preprocessing directive has been encountered. Check for correct spelling.

Example:

```
#i_goofed   // not valid
```

054 *invalid #include directive*

A syntax error has been encountered in a ***#include*** directive.

Example:

```
#include    // no header file
#include stdio.h
```

Both examples are illegal.

055 *not enough parameters given for macro '%s'*

You have not supplied enough parameters to the specified macro.

Example:

```
#define mac(a,b) a+b
int i = mac(123);       // needs 2 parameters
```

056 *not expecting a return value*

The specified function is declared as a ***void*** function. Delete the ***return*** statement, or change the type of the function.

Example:

```
void fun()
{
    return 14;  // not expecting return value
}
```

057 *cannot take address of a bit-field*

The smallest addressable unit is a byte. You cannot take the address of a bit field.

Example:

```
struct S
{   int bits :6;
    int bitfield :10;
};
S var;
void* p = &var.bitfield;    // illegal
```

058 *expression must be constant*

The compiler expects a constant expression. This message can occur during static initialization if you are trying to initialize a non-pointer type with an address expression.

059 *unable to open '%s'*

The file specified in an ***#include*** directive could not be located. Make sure that the file name is spelled correctly, or that the appropriate path for the file is included in the list of paths specified in the **INCLUDE** or **INCLUDE** environment variables or in the "i=" option on the command line.

060 *too many parameters given for macro '%s'*

You have supplied too many parameters for the specified macro. The extra parameters are ignored.

Example:

```
#define mac(a,b) a+b
int i = mac(1,2,3);     // needs 2 parameters
```

061 *cannot use __based or __far16 pointers in this context*

The use of ***__based*** and ***__far16*** pointers is prohibited in ***throw*** expressions and ***catch*** statements.

Example:

```
extern int __based( __segname( "myseg" ) ) *pi;

void bad()
{
    try {
        throw pi;
    } catch( int __far16 *p16 ) {
        *p16 = 87;
    }
}
```

Both the ***throw*** expression and ***catch*** statements cause this error to be diagnosed.

062 *only one type is allowed in declaration specifiers*

Only one type is allowed for the first part of a declaration. A common cause of this message is that there may be a missing semi-colon (';') after a class definition.

Example:

```
class C
{
public:
    C();
}                       // needs ";"

int foo() { return 7; }
```

063 *out of memory*

The compiler has run out of memory to store information about the file being compiled. Try reducing the number of data declarations and or the size of the file being compiled. Do not ***#include*** header files that are not required.

064 *invalid character constant*

This message is issued for an improperly formed character constant.

Example:

```
char c = '12345';
char d = ''';
```

065 *taking address of variable with storage class 'register'*

You can take the address of a ***register*** variable in C++ (but not in ISO/ANSI C). If there is a chance that the source will be compiled using a C compiler, change the storage class from ***register*** to ***auto.***

Example:

```
extern int foo( char* );
int bar()
{
    register char c = 'c';
    return foo( &c );
}
```

066 *'delete' expression size is not allowed*

The C++ language has evolved to the point where the ***delete*** expression size is no longer required for a correct deletion of an array.

Example:

```
void fn( unsigned n, char *p ) {
    delete [n] p;
}
```

067 *ending " missing for string literal*

The compiler did not find a second double quote to end the string literal.

Example:

```
char *a = "no_ending_quote;
```

068 *invalid option*

The specified option is not recognized by the compiler.

069 *invalid optimization option*

The specified option is an unrecognized optimization option.

070 *invalid memory model*

Memory model option must be one of "ms", "mm", "mc", "ml", "mh" or "mf" which selects the Small, Medium, Compact, Large, Huge or Flat memory model.

071 *expression must be integral*

An integral expression is required.

Example:

```
int foo( int a, float b, int *p )
{
    switch( a ) {
      case 1.3:         // must be integral
        return p[b];    // index not integer
      case 2:
        b <<= 2;        // can only shift integers
      default:
        return b;
    }
}
```

072 *expression must be arithmetic*

Arithmetic operations, such as "/" and "*", require arithmetic operands unless the operation has been overloaded or unless the operands can be converted to arithmetic operands.

Example:

```
class C
{
public:
    int c;
};
C cv;
int i = cv / 2;
```

073 *statement required after label*

The C language definition requires a statement following a label. You can use a null statement which consists of just a semicolon (";").

Example:

```
extern int bar( int );
void foo( int a )
{
    if( a ) goto ending;
    bar( a );
ending:
    // needs statement following
}
```

074 *statement required after 'do'*

A statement is required between the ***do*** and ***while*** keywords.

075 *statement required after 'case'*

The C language definition requires a statement following a ***case*** label. You can use a null statement which consists of just a semicolon (";").

Example:

```
int foo( int a )
{
    switch( a ) {
      default:
        return 7;
      case 1: // needs statement following
    }
    return 18;
}
```

076 *statement required after 'default'*

The C language definition requires a statement following a ***default*** label. You can use a null statement which consists of just a semicolon (";").

Example:

```
int foo( int a )
{
    switch( a ) {
      case 7:
        return 7;
      default:
        // needs statement following
    }
    return 18;
}
```

077 *missing matching #endif directive*

You are missing a ***#endif*** to terminate a ***#if, #ifdef*** or ***#ifndef*** preprocessing directive.

Example:

```
#if 1
int a;
// needs #endif
```

078 *invalid macro definition, missing)*

The right parenthesis ")" is required for a function-like macro definition.

Example:

```
#define bad_mac( a, b
```

079 *missing) for expansion of '%s' macro*

The compiler encountered end-of-file while collecting up the argument for a function-like macro. A right parenthesis ")" is required to mark the end of the argument(s) for a function-like macro.

Example:

```
#define mac( a, b) a+b
int d = mac( 1, 2
```

080 *%s*

This is a user message generated with the ***#error*** preprocessing directive.

Example:

```
#error my very own error message
```

081 *cannot define an array of functions*

You can have an array of pointers to functions, but not an array of functions.

Example:

```
typedef int TD(float);
TD array[12];
```

082 *function cannot return an array*

A function cannot return an array. You can return a pointer to an array.

Example:

```
typedef int ARR[10];
ARR fun( float );
```

083 *function cannot return a function*

You cannot return a function. You can return a pointer to a function.

Example:

```
typedef int TD();
TD fun( float );
```

084 *function templates can only have type arguments*

A function template argument can only be a generic type (e.g., `template < class T >`). This is a restriction in the C++ language that allows compilers to automatically instantiate functions purely from the argument types of calls.

085 *maximum class size has been exceeded*

The 16-bit compiler limits the size of a ***struct*** or ***union*** to 64K so that the compiler can represent the offset of a member in a 16-bit register. This error also occurs if the size of a structure overflows the size of an ***unsigned*** integer.

Example:

```
struct S
{
    char arr1[ 0xfffe ];
    char arr2[ 0xfffe ];
    char arr3[ 0xfffe ];
    char arr4[ 0xfffffffe ];
};
```

086 *definition of macro '%s' not identical to previous definition*

If a macro is defined more than once, the definitions must be identical. If you want to redefine a macro to have a different definition, you must ***#undef*** it before you can define it with a new definition.

Example:

```
#define CON 123
#define CON 124     // not same as previous
```

087 *initialization of '%S' must be in file scope*

A file scope variable must be initialized in file scope.

Example:

```
void fn()
{
    extern int v = 1;
}
```

088 *default argument for '%S' declared outside of class definition*

Problems can occur with member functions that do not declare all of their default arguments during the class definition. For instance, a copy constructor is declared if a class does not define a copy constructor. If a default argument is added later on to a constructor that makes it a copy constructor, an ambiguity results.

Example:

```
struct S {
    S( S const &, int );
    // S( S const & );  <-- declared by compiler
};
// ambiguity with compiler
// generated copy constructor
// S( S const & );
S::S( S const &, int = 0 )
{
}
```

089 *## must not be at start or end of replacement tokens*

There must be a token on each side of the "##" (token pasting) operator.

Example:

```
#define badmac( a, b ) ## a ## b
```

090 *invalid floating-point constant*

The exponent part of the floating-point constant is not formed correctly.

Example:

```
float f = 123.9E+Q;
```

091 *'sizeof' is not allowed for a bit-field*

The smallest object that you can ask for the size of is a char.

Example:

```
struct S
{   int a;
    int b :10;
} v;
int k = sizeof( v.b );
```

092 *option requires a path*

The specified option is not recognized by the compiler since there was no path after it (i.e., "-i=d:\include;d:\path").

093 *must use 'va_start' macro inside function with variable arguments*

The `va_start` macro is used to setup access to the parameters in a function that takes a variable number of parameters. A function is defined with a variable number of parameters by declaring the last parameter in the function as "...".

Example:

```
#include <stdarg.h>
int foo( int a, int b )
{
    va_list args;
    va_start( args, a );
    va_end( args );
    return b;
}
```

094 ****FATAL*** %s*

A fatal error has been detected during code generation time. The type of error is displayed in the message.

095 *internal compiler error %d*

A bug has been encountered in the compiler. Please report the specified internal compiler error number and any other helpful details about the program being compiled to Watcom so that we can fix the problem.

096 *argument number %d - invalid register in #pragma*

The designated registers cannot hold the value for the parameter.

097 *procedure '%s' has invalid return register in #pragma*

The size of the return register does not match the size of the result returned by the function.

098 *illegal register modified by '%s' #pragma*

For the 16-bit Watcom C/C++ compiler: The BP, CS, DS, and SS registers cannot be modified in small data models. The BP, CS, and SS registers cannot be modified in large data models.

For the 32-bit Watcom C/C++ compiler: The EBP, CS, DS, ES, and SS registers cannot be modified in flat memory models. The EBP, CS, DS, and SS registers cannot be modified in small data models. The EBP, CS, and SS registers cannot be modified in large data models.

099 *file must contain at least one external definition*

Every file must contain at least one global object, (either a data variable or a function). This message is only issued in strict ISO/ANSI mode (-za).

100 *out of macro space*

The compiler ran out of memory for storing macro definitions.

101 *keyboard interrupt detected*

The compile has been aborted with Ctrl/C or Ctrl/Break.

102 *duplicate macro parameter '%s'*

The parameters specified in a macro definition must be unique.

Example:

```
#define badmac( a, b, a ) a ## b
```

103 *unable to open work file: error code = %d*

The compiler tries to open a new work file by the name "__wrkN__.tmp" where N is the digit 0 to 9. This message will be issued if all of those files already exist.

104 *write error on work file: error code = %d*

An error was encountered trying to write information to the work file. The disk could be full.

105 *read error on work file: error code = %d*

An error was encountered trying to read information from the work file.

106 *token too long; truncated*

The token must be less than 510 bytes in length.

107 *filename required on command line*

The name of a file to be compiled must be specified on the command line.

108 *command line contains more than one file to compile*

You have more than one file name specified on the command line to be compiled. The compiler can only compile one file at a time. You can use the Watcom Compile and Link utility to compile multiple files with a single command.

109 *virtual member functions are not allowed in an union*

A union can only be used to overlay the storage of data. The storage of virtual function information (in a safe manner) cannot be done if storage is overlaid.

Example:

```
struct S1{ int f( int ); };
struct S2{ int f( int ); };
union un { S1 s1;
           S2 s2;
           virtual int vf( int );
         };
```

110 *union cannot be used as a base class*

This restriction prevents C++ programmers from viewing a ***union*** as an encapsulation unit. If it is necessary, one can encapsulate the union into a ***class*** and achieve the same effect.

Example:

```
union U { int a; int b; };
class S : public U { int s; };
```

111 *union cannot have a base class*

This restriction prevents C++ programmers from viewing a ***union*** as an encapsulation unit. If it is necessary, one can encapsulate the union into a ***class*** and inherit the base classes normally.

Example:

```
class S { public: int s; };
union U : public S { int a; int b; };
```

112 *cannot inherit an undefined base class '%T'*

The storage requirements for a ***class*** type must be known when inheritance is involved because the layout of the final class depends on knowing the complete contents of all base classes.

Example:

```
class Undefined;
class C : public Undefined {
    int c;
};
```

113 *repeated direct base class will cause ambiguities*

Almost all accesses will be ambiguous. This restriction is useful in catching programming errors. The repeated base class can be encapsulated in another class if the repetition is required.

Example:

```
class Dup
{
    int d;
};
class C : public Dup, public Dup
{
    int c;
};
```

114 *templates may only be declared in file scope*

Currently, templates can only be declared at file scope. This simple restriction was chosen in favour of more freedom with possibly subtle restrictions.

115 *linkages may only be declared in file scope*

A common source of errors for C and C++ result from the use of prototypes inside of functions. This restriction attempts to prevent such errors.

116 *unknown linkage '%s'*

Only the linkages "C" and "C++" are supported by Watcom C++.

Example:

```
extern "APL" void AplFunc( int* );
```

117 *too many storage class specifiers*

This message is a result of duplicating a previous storage class or having a different storage class. You can only have one of the following storage classes, ***extern, static, auto, register,*** or ***typedef.***

Example:

```
extern typedef int (*fn)( void );
```

118 *nameless declaration is not allowed*

A type was used in a declaration but no name was given.

Example:

```
static int;
```

119 *illegal combination of type specifiers*

An incorrect scalar type was found. Either a scalar keyword was repeated or the combination is illegal.

Example:

```
short short x;
short long y;
```

120 *illegal combination of type qualifiers*

A repetition of a type qualifier has been detected. Some compilers may ignore repetitions but strictly speaking it is incorrect code.

Example:

```
const const x;
struct S {
    int virtual virtual fn();
};
```

121 *syntax error*

The C++ compiler was unable to interpret the text starting at the location of the message. The C++ language is sufficiently complicated that it is difficult for a compiler to correct the error itself.

122 *parser stack corrupted*

The C++ parser has detected an internal problem that usually indicates a compiler problem. Please report this directly to Watcom.

123 *template declarations cannot be nested within each other*

Currently, templates can only be declared at file scope. Furthermore, a template declaration must be finished before another template can be declared.

124 *expression is too complicated*

The expression contains too many levels of nested parentheses. Divide the expression up into two or more sub-expressions.

125 *invalid redefinition of the typedef name '%S'*

Redefinition of typedef names is only allowed if you are redefining a typedef name to itself. Any other redefinition is illegal. You should delete the duplicate ***typedef*** definition.

Example:

```
typedef int TD;
typedef float TD;   // illegal
```

126 *class '%T' has already been defined*

This message usually results from the definition of two classes in the same scope. This is illegal regardless of whether the class definitions are identical.

Example:

```
class C {
};
class C {
};
```

127 *'sizeof' is not allowed for an undefined type*

If a type has not been defined, the compiler cannot know how large it is.

Example:

```
class C;
int x = sizeof( C );
```

128 *initializer for variable '%S' cannot be bypassed*

The variable may not be initialized when code is executing at the position indicated in the message. The C++ language places these restrictions to prevent the use of uninitialized variables.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 1:
        int b = 2;
        return b;
      default: // b bypassed
        return b + 5;
    }
}
```

129 *division by zero in a constant expression*

Division by zero is not allowed in a constant expression. The value of the expression cannot be used with this error.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 4 / 0:  // illegal
        return a;
    }
    return a + 2;
}
```

130 *arithmetic overflow in a constant expression*

The multiplication of two integral values cannot be represented. The value of the expression cannot be used with this error.

Example:

```
int foo( int a )
{
    switch( a ) {
      case 0x7FFF * 0x7FFF * 0x7FFF:  // overflow
        return a;
    }
    return a + 2;
}
```

131 *not enough memory to fully optimize procedure '%s'*

The indicated procedure cannot be fully optimized with the amount of memory available. The code generated will still be correct and execute properly. This message is purely informational (i.e., buy more memory).

132 *not enough memory to maintain full peephole*

Certain optimizations benefit from being able to store the entire module in memory during optimization. All functions will be individually optimized but the optimizer will not be able to share code between functions if this message appears. The code generated will still be correct and execute properly. This message is purely informational (i.e., buy more memory).

133 *too many errors: compilation aborted*

The Watcom C++ compiler sets a limit to the number of error messages it will issue. Once the number of messages reaches the limit the above message is issued. This limit can be changed via the "/e" command line option.

134 *too many parm sets*

An extra parameter passing description has been found in the aux pragma text. Only one parameter passing description is allowed.

135 *'friend', 'virtual' or 'inline' modifiers may only be used on functions*

This message indicates that you are trying to declare a strange entity like an ***inline*** variable. These qualifiers can only be used on function declarations and definitions.

136 *more than one calling convention has been specified*

A function cannot have more than one #pragma modifier applied to it. Combine the pragmas into one pragma and apply it once.

137 *pure member function constant must be '0'*

The constant must be changed to '0' in order for the Watcom C++ compiler to accept the pure virtual member function declaration.

Example:

```
struct S {
    virtual int wrong( void ) = 91;
};
```

138 *'based' modifier has been repeated*

A repeated based modifier has been detected. There are no semantics for combining base modifiers so this is not allowed.

Example:

```
char *ptr;
char __based( void ) __based( ptr ) *a;
```

139 *enumeration variable is not assigned a constant from its enumeration*

In C++ (as opposed to C), enums represent values of distinct types. Thus, the compiler will not automatically convert an integer value to an enum type if you are compiling your source in strict ISO/ANSI C++ mode. If you have extensions enabled, this message is treated as a warning.

Example:

```
enum Days { sun, mod, tues, wed, thur, fri, sat };
enum Days day = 2;
```

140 *bit-field declaration cannot have a storage class specifier*

Bit-fields (along with most members) cannot have storage class specifiers in their declaration. Remove the storage class specifier to correct the code.

Example:

```
class C
{
public:
    extern unsigned bitf :10;
};
```

141 *bit-field declaration must have a base type specified*

A bit-field cannot make use of a default integer type. Specify the type ***int*** to correct the code.

Example:

```
class C
{
public:
    bitf :10;
};
```

142 *illegal qualification of a bit-field declaration*

A bit-field can only be declared ***const*** or ***volatile.*** Qualifications like ***friend*** are not allowed.

Example:

```
struct S {
    friend int bit1 :10;
    inline int bit2 :10;
    virtual int bit3 :10;
};
```

All three declarations of bit-fields are illegal.

143 *duplicate base qualifier*

The compiler has found a repetition of base qualifiers like ***protected*** or ***virtual.***

Example:

```
struct Base { int b; };
struct Derived : public public Base { int d; };
```

144 *only one access specifier is allowed*

The compiler has found more than one access specifier for a base class. Since the compiler cannot choose one over the other, remove the unwanted access specifier to correct the code.

Example:

```
struct Base { int b; };
struct Derived : public protected Base { int d; };
```

145 *unexpected type qualifier found*

Type specifiers cannot have ***const*** or ***volatile*** qualifiers. This shows up in ***new*** expressions because one cannot allocate a ***const*** object.

146 *unexpected storage class specifier found*

Type specifiers cannot have ***auto*** or ***static*** storage class specifiers. This shows up in ***new*** expressions because one cannot allocate a ***static*** object.

147 *access to '%S' is not allowed because it is ambiguous*

There are two ways that this error can show up in C++ code. The first way a member can be ambiguous is that the same name can be used in two different classes. If these classes are combined with multiple inheritance, accesses of the name will be ambiguous.

Example:

```
struct S1 { int s; };
struct S2 { int s; };
struct Der : public S1, public S2
{
    void foo() { s = 2; };  // s is ambiguous
};
```

The second way a member can be ambiguous involves multiple inheritance. If a class is inherited non-virtually by two different classes which then get combined with multiple inheritance, an access of the member is faced with deciding which copy of the member is intended. Use the '::' operator to clarify what member is being accessed or access the member with a different class pointer or reference.

Example:

```
struct Top { int t; };
struct Mid : public Top { int m; };
struct Bot : public Top, public Mid
{
    void foo() { t = 2; };  // t is ambiguous
};
```

148 *access to private member '%S' is not allowed*

The indicated member is being accessed by an expression that does not have permission to access private members of the class.

Example:

```
struct Top { int t; };
class Bot : private Top
{
    int foo() { return t; };  // t is private
};
Bot b;
int k = b.foo();    // foo is private
```

149 *access to protected member '%S' is not allowed*

The indicated member is being accessed by an expression that does not have permission to access protected members of the class. The compiler also requires that ***protected*** members be accessed through a derived class to ensure that an unrelated base class cannot be quietly modified. This is a fairly recent change to the C++ language that may cause Watcom C++ to not accept older C++ code. See Section 11.5 in the ARM for a discussion of protected access.

Example:

```
struct Top { int t; };
struct Mid : public Top { int m; };
class Bot : protected Mid
{
protected:
    // t cannot be accessed
    int foo() { return t; };
};
Bot b;
int k = b.foo(); // foo is protected
```

150 *operation does not allow both operands to be pointers*

There may be a missing indirection in the code exhibiting this error. An example of this error is adding two pointers.

Example:

```
void fn()
{
    char *p, *q;

    p += q;
}
```

151 *operand is neither a pointer nor an arithmetic type*

An example of this error is incrementing a class that does not have any overloaded operators.

Example:

```
struct S { } x;
void fn()
{
    ++x;
}
```

152 *left operand is neither a pointer nor an arithmetic type*

An example of this error is trying to add 1 to a class that does not have any overloaded operators.

Example:

```
struct S { } x;
void fn()
{
    x = x + 1;
}
```

153 *right operand is neither a pointer nor an arithmetic type*

An example of this error is trying to add 1 to a class that does not have any overloaded operators.

Example:

```
struct S { } x;
void fn()
{
    x = 1 + x;
}
```

154 *cannot subtract a pointer from an arithmetic operand*

The subtract operands are probably in the wrong order.

Example:

```
int fn( char *p )
{
    return( 10 - p );
}
```

155 *left expression must be arithmetic*

Certain operations like multiplication require both operands to be of arithmetic types.

Example:

```
struct S { } x;
void fn()
{
    x = x * 1;
}
```

156 *right expression must be arithmetic*

Certain operations like multiplication require both operands to be of arithmetic types.

Example:

```
struct S { } x;
void fn()
{
    x = 1 * x;
}
```

157 *left expression must be integral*

Certain operators like the bit manipulation operators require both operands to be of integral types.

Example:

```
struct S { } x;
void fn()
{
    x = x ^ 1;
}
```

158 *right expression must be integral*

Certain operators like the bit manipulation operators require both operands to be of integral types.

Example:

```
struct S { } x;
void fn()
{
    x = 1 ^ x;
}
```

159 *cannot assign a pointer value to an arithmetic item*

The pointer value must be cast to the desired type before the assignment takes place.

Example:

```
void fn( char *p )
{
    int a;

    a = p;
}
```

160 *attempt to destruct a far object when data model is near*

Destructors cannot be applied to objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
    ~Obj();
};

Obj far obj;
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

161 *attempt to call member function for far object when the data model is near*

Member functions cannot be called for objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
    int foo();
};

Obj far obj;
int integer = obj.foo();
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

162 *template type argument cannot have a default argument*

Currently, the C++ language does not allow template type arguments to have a default type. There is neither syntax to express the notion nor semantics defined for its meaning.

163 *attempt to 'delete' a far object when the data model is near*

delete cannot be used to deallocate objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
};

void foo( Obj far *p )
{
    delete p;
}
```

The second last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

164 *first operand is not a class, struct or union*

The ***offsetof*** operation can only be performed on a type that can have members. It is meaningless for any other type.

Example:

```
#include <stddef.h>

int fn( void )
{
    return offsetof( double, sign );
}
```

165 *syntax error; class template cannot be processed*

The class template contains unbalanced braces. The class definition cannot be processed in this form.

166 *cannot convert right pointer to type of left operand*

The C++ language will not allow the implicit conversion of unrelated class pointers. An explicit cast is required.

Example:

```
class C1;
class C2;

void fun( C1* pc1, C2* pc2 )
{
    pc2 = pc1;
}
```

167 *left operand must be an 'lvalue'*

The left operand must be an expression that is valid on the left side of an assignment. Examples of incorrect lvalues include constants and the results of most operators.

Example:

```
int i, j;
void fn()
{
    ( i - 1 ) = j;
    1 = j;
}
```

168 *static data members are not allowed in an union*

A union should only be used to organize memory in C++. Enclose the union in a class if you need a static data member associated with the union.

Example:

```
union U
{
    static int a;
    int b;
    int c;
};
```

169 *invalid storage class for a member*

A class member cannot be declared with ***auto, register,*** or ***extern*** storage class.

Example:

```
class C
{
    auto int a;     // cannot specify auto
};
```

170 *declaration is too complicated*

The declaration contains too many declarators (i.e., pointer, array, and function types). Break up the declaration into a series of typedefs ending in a final declaration.

Example:

```
int ************p;
```

Example:

```
// transform this to ...
typedef int ****PD1;
typedef PD1 ****PD2;
PD2 ****p;
```

171 *exception declaration is too complicated*

The exception declaration contains too many declarators (i.e., pointer, array, and function types). Break up the declaration into a series of typedefs ending in a final declaration.

172 *floating-point constant too large to represent*

The Watcom C++ compiler cannot represent the floating-point constant because the magnitude of the positive exponent is too large.

Example:

```
float f = 1.2e78965;
```

173 *floating-point constant too small to represent*

The Watcom C++ compiler cannot represent the floating-point constant because the magnitude of the negative exponent is too large.

Example:

```
float f = 1.2e-78965;
```

174 *class template '%S' cannot be overloaded*

A class template name must be unique across the entire C++ program. Furthermore, a class template cannot coexist with another class template of the same name.

175 *range of enum constants cannot be represented*

If one integral type cannot be chosen to represent all values of an enumeration, the values cannot be used reliably in the generated code. Shrink the range of enumerator values used in the ***enum*** declaration.

Example:

```
enum E
{   e1 = 0xFFFFFFFF
,   e2 = -1
};
```

176 *'%S' cannot be in the same scope as a class template*

A class template name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the class template.

177 *invalid storage class in file scope*

A declaration in file scope cannot have a storage class of ***auto*** or ***register.***

Example:

```
auto int a;
```

178 *const object must be initialized*

Constant objects cannot be modified so they must be initialized before use.

Example:

```
const int a;
```

179 *declaration cannot be in the same scope as class template '%S'*

A class template name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the class template.

180 *template arguments must be named*

A member function of a template class cannot be defined outside the class declaration unless all template arguments have been named.

181 *class template '%S' is already defined*

A class template cannot have its definition repeated regardless of whether it is identical to the previous definition.

182 *invalid storage class for an argument*

An argument declaration cannot have a storage class of ***extern, static,*** or ***typedef.***

Example:

```
int foo( extern int a )
{
    return a;
}
```

183 *unions cannot have members with constructors*

A union should only be used to organize memory in C++. Allowing union members to have constructors would mean that the same piece of memory could be constructed twice.

Example:

```
class C
{ C();
};
union U
{
    int a;
    C c;        // has constructor
};
```

184 *statement is too complicated*

The statement contains too many nested constructs. Break up the statement into multiple statements.

185 *'%s' is not the name of a class or namespace*

The right hand operand of a '::' operator turned out not to reference a class type or namespace. Because the name is followed by another '::', it must name a class or namespace.

186 *attempt to modify a constant value*

Modification of a constant value is not allowed. If you must force this to work, take the address and cast away the constant nature of the type.

Example:

```
static int const con = 12;
void foo()
{
    con = 13;           // error
    *(int*)&con = 13;   // ok
}
```

187 *'offsetof' is not allowed for a bit-field*

A bit-field cannot have a simple offset so it cannot be referenced in an ***offsetof*** expression.

Example:

```
#include <stddef.h>
struct S
{
    unsigned b1 :10;
    unsigned b2 :15;
    unsigned b3 :11;
};
int k = offsetof( S, b2 );
```

188 *base class is inherited with private access*

This warning indicates that the base class was originally declared as a ***class*** as opposed to a ***struct.*** Furthermore, no access was specified so the base class defaults to ***private*** inheritance. Add the ***private*** or ***public*** access specifier to prevent this message depending on the intended access.

189 *overloaded function cannot be selected for arguments used in call*

Either conversions were not possible for an argument to the function or a function with the right number of arguments was not available.

Example:

```
class C1;
class C2;
int foo( C1* );
int foo( C2* );
int k = foo( 5 );
```

190 *base operator operands must be " __segment :> pointer "*

The base operator (:>) requires the left operand to be of type __segment and the right operand to be a pointer.

Example:

```
char _based( void ) *pcb;
char __far *pcf = pcb;      // needs :> operator
```

Examples of typical uses are as follows:

Example:

```
const __segment mySegAbs = 0x4000;
char __based( void ) *c_bv = 24;
char __far *c_fp_1 = mySegAbs :> c_bv;
char __far *c_fp_2 = __segname( "_DATA" ) :> c_bv;
```

191 *expression must be a pointer or a zero constant*

In a conditional expression, if one side of the ':' is a pointer then the other side must also be a pointer or a zero constant.

Example:

```
extern int a;
int *p = ( a > 7 ) ? &a : 12;
```

192 *left expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:

```
void *p;
void *q = p + 2;
```

193 *right expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:

```
void *p;
void *q = 2 + p;
```

194 *expression pointer type cannot be incremented or decremented*

The expression requires that the scaling size of the pointer be known. Pointers to functions, arrays of unknown size, or ***void*** cannot be incremented because there is no size defined for functions, arrays of unknown size, or ***void.***

Example:

```
void *p;
void *q = ++p;
```

195 *'sizeof' is not allowed for a function*

A function has no size defined for it by the C++ language specification.

Example:

```
typedef int FT( int );

unsigned y = sizeof( FT );
```

196 *'sizeof' is not allowed for a type 'void'*

The type ***void*** has no size defined for it by the C++ language specification.

Example:

```
void *p;
unsigned size = sizeof( *p );
```

197 *type cannot be defined in this context*

A type cannot be defined in certain contexts. For example, a new type cannot be defined in an argument list, a ***new*** expression, a conversion function identifier, or a catch handler.

Example:

```
extern int goop();
int foo()
{
    try {
        return goop();
    } catch( struct S { int s; } ) {
        return 2;
    }
}
```

198 *expression cannot be used as a class template parameter*

The compiler has to be able to compare expressions during compilation so this limits the complexity of expressions that can be used for template parameters. The only types of expressions that can be used for template parameters are constant integral expressions and addresses. Any symbols must have external linkage or must be static class members.

199 *premature end-of-file encountered during compilation*

The compiler expects more source code at this point. This can be due to missing parentheses (')') or missing closing braces ('}').

200 *duplicate case value '%s' after conversion to type of switch expression*

A duplicate ***case*** value has been found. Keep in mind that all case values must be converted to the type of the switch expression. Constants that may be different initially may convert to the same value.

Example:

```
enum E { e1, e2 };
void foo( short a )
{
    switch( a ) {
      case 1:
      case 0x10001:    // converts to 1 as short
        break;
    }
}
```

201 *declaration statement follows an if statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    if( a )
        int b = 14;
}
```

202 *declaration statement follows an else statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    if( a )
        int c = 15;
    else
        int b = 14;
}
```

203 *declaration statement follows a switch statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    switch( a )
        int b = 14;
}
```

204 *'this' pointer is not defined*

The ***this*** value can only be used from within non-static member functions.

Example:

```
void *fn()
{
    return this;
}
```

205 *declaration statement cannot follow a while statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    while( a )
        int b = 14;
}
```

206 *declaration statement cannot follow a do statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended.

Example:

```
void foo( int a )
{
    do
        int b = 14;
    while( a );
}
```

207 *declaration statement cannot follow a for statement*

There are implicit scopes created for most control structures. Because of this, no code can access any of the names declared in the declaration. Although the code is legal it may not be what the programmer intended. A ***for*** loop with an initial declaration is allowed to be used within another ***for*** loop, so this code is legal C++:

Example:

```
void fn( int **a )
{
    for( int i = 0; i < 10; ++i )
        for( int j = 0; j < 10; ++j )
            a[i][j] = i + j;
}
```

The following example, however, illustrates a potentially erroneous situation.

Example:

```
void foo( int a )
{
    for( ; a<10; )
        int b = 14;
}
```

208 *pointer to virtual base class converted to pointer to derived class*

Since the relative position of a virtual base can change through repeated derivations, this conversion is very dangerous. All C++ translators must report an error for this type of conversion.

Example:

```
struct VBase { int v; };
struct Der : virtual public VBase { int   d; };
extern VBase *pv;
Der *pd = (Der *)pv;
```

209 *cannot use far pointer in this context*

Only near pointers can be thrown when the data memory model is near.

Example:

```
extern int __far *p;
void foo()
{
    throw p;
}
```

When the small memory model (-ms switch) is selected, the ***throw*** expression is diagnosed as erroneous. Similarly, only near pointers can be specified in ***catch*** statements when the data memory model is near.

210 *returning reference to function argument or to auto or register variable*

The storage for the automatic variable will be destroyed immediately upon function return. Returning a reference effectively allows the caller to modify storage which does not exist.

Example:

```
class C
{
    char *p;
public:
    C();
    ~C();
};

C& foo()
{
    C auto_var;
    return auto_var;    // not allowed
}
```

211 *#pragma attributes for '%S' may be inconsistent*

A pragma attribute was changed to a value which matches neither the current default not the previous value for that attribute. A warning is issued since this usually indicates an attribute is being set twice (or more) in an inconsistent way. The warning can also occur when the default attribute is changed between two pragmas for the same object.

212 *function arguments cannot be of type 'void'*

Having more than one ***void*** argument is not allowed. The special case of one ***void*** argument indicates that the function accepts no parameters.

Example:

```
void fn1( void )                // OK
{
}
void fn2( void, void, void )    // Error!
{
}
```

213 *class template requires more parameters for instantiation*

The class template instantiation has too few parameters supplied so the class cannot be instantiated properly.

214 *class template requires less parameters for instantiation*

The class template instantiation has too many parameters supplied so the class cannot be instantiated properly.

215 *no declared 'operator new' has arguments that match*

An ***operator new*** could not be found to match the ***new*** expression. Supply the correct arguments for special ***operator new*** functions that are defined with the placement syntax.

Example:

```
#include <stddef.h>

struct S {
    void *operator new( size_t, char );
};

void fn()
{
    S *p = new ('a') S;
}
```

216 *wide character string concatenated with a simple character string*

There are no semantics defined for combining a wide character string with a simple character string. To correct the problem, make the simple character string a wide character string by prefixing it with a ***L.***

Example:

```
char *p = "1234" L"5678";
```

217 *'offsetof' is not allowed for a static member*

A ***static*** member does not have an offset like simple data members. If this is required, use the address of the ***static*** member.

Example:

```
#include <stddef.h>
class C
{
public:
    static int stat;
    int memb;
};

int size_1 = offsetof( C, stat );   // not allowed
int size_2 = offsetof( C, memb );   // ok
```

218 *cannot define an array of 'void'*

Since the ***void*** type has no size and there are no values of ***void*** type, one cannot declare an array of ***void.***

Example:

```
void array[24];
```

219 *cannot define an array of references*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating an array of references is currently not allowed in the C++ language.

Example:

```
int& array[24];
```

220 *cannot define a reference to 'void'*

One cannot create a reference to a ***void*** because there can be no ***void*** variables to supply for initializing the reference.

Example:

```
void& ref;
```

221 *cannot define a reference to another reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a reference to another reference is currently not allowed in the C++ language.

Example:

```
int & & ref;
```

222 *cannot define a pointer to a reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a pointer to a reference is currently not allowed in the C++ language.

Example:

```
char& *ptr;
```

223 *cannot initialize array with 'operator new'*

The initialization of arrays created with ***operator new*** can only be done with default constructors. The capability of using another constructor with arguments is currently not allowed in the C++ language.

Example:

```
struct S
{
    S( int );
};
S *p = new S[10] ( 12 );
```

224 *'%N' is a variable of type 'void'*

A variable cannot be of type ***void.*** The ***void*** type can only be used in restricted circumstances because it has no size. For instance, a function returning ***void*** means that it does not return any value. A pointer to ***void*** is used as a generic pointer but it cannot be dereferenced.

225 *cannot define a member pointer to a reference*

References are not objects, they are simply a way of creating an efficient alias to another name. Creating a member pointer to a reference is currently not allowed in the C++ language.

Example:

```
struct S
{
    S();
    int &ref;
};

int& S::* p;
```

226 *function '%S' is not distinct*

The function being declared is not distinct enough from the other functions of the same name. This means that all function overloads involving the function's argument types will be ambiguous.

Example:

```
struct S {
    int s;
};
extern int foo( S* );
extern int foo( S* const ); // not distinct enough
```

227 *overloaded function is ambiguous for arguments used in call*

The compiler could not find an unambiguous choice for the function being called.

Example:

```
extern int foo( char );
extern int foo( short );
int k = foo( 4 );
```

228 *declared 'operator new' is ambiguous for arguments used*

The compiler could not find an unambiguous choice for ***operator new.***

Example:

```
#include <stdlib.h>
struct Der
{
    int s[2];
    void* operator new( size_t, char );
    void* operator new( size_t, short );
};
Der *p = new(10) Der;
```

229 *function '%S' has already been defined*

The function being defined has already been defined elsewhere. Even if the two function bodies are identical, there must be only one definition for a particular function.

Example:

```
int foo( int s ) { return s; }
int foo( int s ) { return s; }  // illegal
```

230 *expression on left is an array*

The array expression is being used in a context where only pointers are allowed.

Example:

```
void fn( void *p )
{
    int a[10];

    a = 0;
    a = p;
    a++;
}
```

231 *user-defined conversion has a return type*

A user-defined conversion cannot be declared with a return type. The "return type" of the user-defined conversion is implicit in the name of the user-defined conversion.

Example:

```
struct S {
    int operator int(); // cannot have return type
};
```

232 *user-defined conversion must be a function*

The operator name describing a user-defined conversion can only be used to designate functions.

Example:

```
// operator char can only be a function
int operator char = 9;
```

233 *user-defined conversion has an argument list*

A user-defined conversion cannot have an argument list. Since user-defined conversions can only be non-static member functions, they have an implicit ***this*** argument.

Example:

```
struct S {
    operator int( S& ); // cannot have arguments
};
```

234 *destructor cannot have a return type*

A destructor cannot have a return type (even ***void***). The destructor is a special member function that is not required to be identical in form to all other member functions. This allows different implementations to have different uses for any return values.

Example:

```
struct S {
    void* ~S();
};
```

235 *destructor must be a function*

The tilde ('~') style of name is reserved for declaring destructor functions. Variable names cannot make use of the destructor style of names.

Example:

```
struct S {
    int ~S; // illegal
};
```

236 *destructor has an argument list*

A destructor cannot have an argument list. Since destructors can only be non-static member functions, they have an implicit ***this*** argument.

Example:

```
struct S {
    ~S( S& );
};
```

237 *'%N' must be a function*

The ***operator*** style of name is reserved for declaring operator functions. Variable names cannot make use of the ***operator*** style of names.

Example:

```
struct S {
    int operator+;  // illegal
};
```

238 *'%N' is not a function*

The compiler has detected what looks like a function body. The message is a result of not finding a function being declared. This can happen in many ways, such as dropping the ':' before defining base classes, or dropping the '=' before initializing a structure via a braced initializer.

Example:

```
struct D B { int i; };
```

239 *nested type 'class %s' has not been declared*

A nested class has not been found but is required by the use of repeated '::' operators. The construct "A::B::C" requires that 'A' be a class type, and 'B' be a nested class within the scope of 'A'.

Example:

```
struct B {
    static int b;
};
struct A : public B {
};
int A::B::b = 2;    // B not nested in A
```

The preceding example is illegal; the following is legal

Example:

```
struct A {
    struct B {
        static int b;
    };
};
int A::B::b = 2;    // B nested in A
```

240 *'enum %s' has not been declared*

An elaborated reference to an ***enum*** could not be satisfied. All enclosing scopes have been searched for an ***enum*** name. Visible variable declarations do not affect the search.

Example:

```
struct D {
    int i;
    enum E { e1, e2, e3 };
};
enum E enum_var;    // E not visible
```

241 *'class %s' has not been declared*

The construct "A::B::C" requires that 'A' be a class type, and 'B' be a nested class within the scope of 'A'. The reference to 'A' could not be satisfied. All enclosing scopes have been searched for a ***class*** name. Visible variable declarations do not affect the search.

Example:

```
struct A{ int a; };

int b;
int c = B::A::b;
```

242 *only one initializer argument allowed*

The comma (’,’) in a function like cast is treated like an argument list comma (’,’). If a comma expression is desired, use parentheses to enclose the comma expression.

Example:

```
void fn()
{
    int a;

    a = int( 1, 2 );            // Error!
    a = int( ( 1, 2 ) );        // OK
}
```

243 *default arguments are not part of a function’s type*

This message indicates that a declaration has been found that requires default arguments to be part of a function’s type. Either declaring a function ***typedef*** or a pointer to a function with default arguments are examples of incorrect declarations.

Example:

```
typedef int TD( int, int a = 14 );
int (*p)( int, int a = 14 ) = 0;
```

244 *missing default arguments*

Gaps in a succession of default arguments are not allowed in the C++ language.

Example:

```
void fn( int = 1, int, int = 3 );
```

245 *overloaded operator cannot have default arguments*

Preventing overloaded operators from having default arguments enforces the property that binary operators will only be called from a use of a binary operator. Allowing default arguments would allow a binary ***operator +*** to function as a unary ***operator +.***

Example:

```
class C
{
public:
    C operator +( int a = 10 );
};
```

246 *left expression is not a pointer to a constant object*

One cannot assign a pointer to a constant type to a pointer to a non-constant type. This would allow a constant object to be modified via the non-constant pointer. Use a cast if this is absolutely necessary.

Example:

```
char* fun( const char* p )
{
    char* q;
    q = p;
    return q;
}
```

247 *cannot redefine default argument for '%S'*

Default arguments can only be defined once in a program regardless of whether the value of the default argument is identical.

Example:

```
static int foo( int a = 10 );
static int foo( int a = 10 )
{
    return a+a;
}
```

248 *using default arguments would be overload ambiguous with '%S'*

The declaration declares enough default arguments that the function is indistinguishable from another function of the same name.

Example:

```
void fn( int );
void fn( int, int = 1 );
```

Calling the function 'fn' with one argument is ambiguous because it could match either the first 'fn' without any default arguments or the second 'fn' with a default argument applied.

249 *using default arguments would be overload ambiguous with '%S' using default arguments*

The declaration declares enough default arguments that the function is indistinguishable from another function of the same name with default arguments.

Example:

```
void fn( int, int = 1 );
void fn( int, char = 'a' );
```

Calling the function 'fn' with one argument is ambiguous because it could match either the first 'fn' with a default argument or the second 'fn' with a default argument applied.

250 *missing default argument for '%S'*

In C++, one is allowed to add default arguments to the right hand arguments of a function declaration in successive declarations. The message indicates that the declaration is only valid if there was a default argument previously declared for the next argument.

Example:

```
void fn1( int    , int     );
void fn1( int    , int = 3 );
void fn1( int = 2, int     );   // OK

void fn2( int    , int     );
void fn2( int = 2, int     );   // Error!
```

251 *enum references must have an identifier*

There is no way to reference an anonymous ***enum.*** If all enums are named, the cause of this message is most likely a missing identifier.

Example:

```
enum   { X, Y, Z };     // anonymous enum
void fn()
{
    enum *p;
}
```

252 *class declaration has not been seen for '~%s'*

A destructor has been used in a context where its class is not visible.

Example:

```
class C;

void fun( C* p )
{
    p->~S();
}
```

253 *'::' qualifier cannot be used in this context*

Qualified identifiers in a class context are allowed for declaring ***friend*** member functions. The Watcom C++ compiler also allows code that is qualified with its own class so that declarations can be moved in and out of class definitions easily.

Example:

```
struct N {
    void bar();
};
struct S {
    void S::foo() { // OK
    }
    void N::bar() { // error
    }
};
```

254 *'%S' has not been declared as a member*

In a definition of a class member, the indicated declaration must already have been declared when the class was defined.

Example:

```
class C
{
public:
    int c;
    int goop();
};
int C::x = 1;
C::not_decled() { }
```

255 *default argument expression cannot use function argument '%S'*

Default arguments must be evaluated at each call. Since the order of evaluation for arguments is undefined, a compiler must diagnose all default arguments that depend on other arguments.

Example:

```
void goop( int d )
{
    struct S {
        // cannot access "d"
        int foo( int c, int b = d )
            {
                return b + c;
            };
    };
}
```

256 *default argument expression cannot use local variable '%S'*

Default arguments must be evaluated at each call. Since a local variable is not always available in all contexts (e.g., file scope initializers), a compiler must diagnose all default arguments that depend on local variables.

Example:

```
void goop( void )
{
    int a;
    struct S {
        // cannot access "a"
        int foo( int c, int b = a )
            {
                return b + c;
            };
    };
}
```

257 *access declarations may only be 'public' or 'protected'*

Access declarations are used to increase access. A ***private*** access declaration is useless because there is no access level for which ***private*** is an increase in access.

Example:

```
class Base
{
    int pri;
protected:
    int pro;
public:
    int pub;
};
class Derived : public Base
{
    private: Base::pri;
};
```

258 *cannot declare both a function and variable of the same name ('%N')*

Functions can be overloaded in C++ but they cannot be overloaded in the presence of a variable of the same name. Likewise, one cannot declare a variable in the same scope as a set of overloaded functions of the same name.

Example:

```
int foo();
int foo;
struct S {
    int bad();
    int bad;
};
```

259 *class in access declaration ('%T') must be a direct base class*

Access declarations can only be applied to direct (immediate) base classes.

Example:

```
struct B {
    int f;
};
struct C : B {
    int g;
};
struct D : private C {
    B::f;
};
```

In the above example, "C" is a direct base class of "D" and "B" is a direct base class of "C", but "B" is not a direct base class of "D".

260 *overloaded functions ('%N') do not have the same access*

If an access declaration is referencing a set of overloaded functions, then they all must have the same access. This is due to the lack of a type in an access declaration.

Example:

```
class C
{
    static int foo( int );     // private
public:
    static int foo( float );   // public
};

class B : private C
{
public: C::foo;
};
```

261 *cannot grant access to '%N'*

A derived class cannot change the access of a base class member with an access declaration. The access declaration can only be used to restore access changed by inheritance.

Example:

```
class Base
{
public:
    int pub;
protected:
    int pro;
};
class Der : private Base
{
    public: Base::pub;       // ok
    public: Base::pro;       // changes access
};
```

262 *cannot reduce access to '%N'*

A derived class cannot change the access of a base class member with an access declaration. The access declaration can only be used to restore access changed by inheritance.

Example:

```
class Base
{
public:
    int pub;
protected:
    int pro;
};
class Der : public Base
{
    protected: Base::pub;       // changes access
    protected: Base::pro;       // ok
};
```

263 *nested class '%N' has not been defined*

The current state of the C++ language supports nested types. Unfortunately, this means that some working C code will not work unchanged.

Example:

```
struct S {
    struct T;
    T *link;
};
```

In the above example, the class "T" will be reported as not being defined by the end of the class declaration. The code can be corrected in the following manner.

Example:

```
struct S {
    struct T;
    T *link;
    struct T {
    };
};
```

264 *user-defined conversion must be a non-static member function*

A user-defined conversion is a special member function that allows the class to be converted implicitly (or explicitly) to an arbitrary type. In order to do this, it must have access to an instance of the class so it is restricted to being a non-static member function.

Example:

```
struct S
{
    static operator int();
};
```

265 *destructor must be a non-static member function*

A destructor is a special member function that will perform cleanup on a class before the storage for the class will be released. In order to do this, it must have access to an instance of the class so it is restricted to being a non-static member function.

Example:

```
struct S
{
    static ~S();
};
```

266 *'%N' must be a non-static member function*

The operator function in the message is restricted to being a non-static member function. This usually means that the operator function is treated in a special manner by the compiler.

Example:

```
class C
{
public:
    static operator =( C&, int );
};
```

267 *'%N' must have one argument*

The operator function in the message is only allowed to have one argument. An operator like ***operator*** **~** is one such example because it represents a unary operator.

Example:

```
class C
{
public: int c;
};
C& operator~( const C&, int );
```

268 *'%N' must have two arguments*

The operator function in the message must have two arguments. An operator like ***operator*** **+=** is one such example because it represents a binary operator.

Example:

```
class C
{
public: int c;
};
C& operator += ( const C& );
```

269 *'%N' must have either one argument or two arguments*

The operator function in the message must have either one argument or two arguments. An operator like ***operator +*** is one such example because it represents either a unary or a binary operator.

Example:

```
class C
{
public: int c;
};
C& operator+( const C&, int, float );
```

270 *'%N' must have at least one argument*

The ***operator new*** and ***operator new []*** member functions must have at least one argument for the size of the allocation. After that, any arguments are up to the programmer. The extra arguments can be supplied in a ***new*** expression via the placement syntax.

Example:

```
#include <stddef.h>

struct S {
    void * operator new( size_t, char );
};

void fn()
{
    S *p = new ('a') S;
}
```

271 *'%N' must have a return type of 'void'*

The C++ language requires that ***operator delete*** and ***operator delete []*** have a return type of ***void.***

Example:

```
class C
{
public:
    int c;
    C* operator delete( void* );
    C* operator delete []( void* );
};
```

272 *'%N' must have a return type of 'void *'*

The C++ language requires that both ***operator new*** and ***operator new []*** have a return type of `void *.`

Example:

```
#include <stddef.h>
class C
{
public:
    int c;
    C* operator new( size_t size );
    C* operator new []( size_t size );
};
```

273 *the first argument of '%N' must be of type 'size_t'*

The C++ language requires that the first argument for ***operator new*** and ***operator new []*** be of the type "size_t". The definition for "size_t" can be included by using the standard header file <stddef.h>.

Example:

```
void *operator new( int size );
void *operator new( double size, char c );
void *operator new []( int size );
void *operator new []( double size, char c );
```

274 *the first argument of '%N' must be 'void *'*

The C++ language requires that the first argument for ***operator delete*** and ***operator delete []*** be a `void *`.

Example:

```
class C;
void operator delete( C* );
void operator delete []( C* );
```

275 *the second argument of '%N' must be of type 'size_t'*

The C++ language requires that the second argument for ***operator delete*** and ***operator delete []*** be of type "size_t". The two argument form of ***operator delete*** and ***operator delete []*** is optional and it can only be present inside of a class declaration. The definition for "size_t" can be included by using the standard header file <stddef.h>.

Example:

```
struct S {
    void operator delete( void *, char );
    void operator delete []( void *, char );
};
```

276 *the second argument of 'operator ++' or 'operator --' must be 'int'*

The C++ language requires that the second argument for ***operator ++*** be ***int.*** The two argument form of ***operator ++*** is used to overload the postfix operator "++". The postfix operator "--" can be overloaded similarly.

Example:

```
class C {
public:
    long cv;
};
C& operator ++( C&, unsigned );
```

277 *return type of '%S' must allow the '->' operator to be applied*

This restriction is a result of the transformation that the compiler performs when the ***operator ->*** is overloaded. The transformation involves transforming the expression to invoke the operator with "->" applied to the result of ***operator ->.***

Example:

```
struct S {
    int a;
    S *operator ->();
};

void fn( S &q )
{
    q->a = 1; // becomes (q.operator ->())->a = 1;
}
```

278 *'%N' must take at least one argument of a class/enum or a reference to a class/enum*

Overloaded operators can only be defined for classes and enumerations. At least one argument, must be a class or an enum type in order for the C++ compiler to distinguish the operator from the built-in operators.

Example:

```
class C {
public:
    long cv;
};
C& operator ++( unsigned, int );
```

279 *too many initializers*

The compiler has detected extra initializers.

Example:

```
int a[3] = { 1, 2, 3, 4 };
```

280 *too many initializers for character string*

A string literal used in an initialization of a character array is viewed as providing the terminating null character. If the number of array elements isn't enough to accept the terminating character, this message is output.

Example:

```
char ac[3] = "abc";
```

281 *expecting '%s' but found expression*

This message is output when some bracing or punctuation is expected but an expression was encountered.

Example:

```
int b[3] = 3;
```

282 *anonymous struct/union member '%N' cannot be declared in this class*

An anonymous member cannot be declared with the same name as its containing class.

Example:

```
struct S {
    union {
        int S;          // Error!
        char b;
    };
};
```

283 *unexpected '%s' during initialization*

This message is output when some unexpected bracing or punctuation is encountered during initialization.

Example:

```
int e = { { 1 };
```

284 *nested type '%N' cannot be declared in this class*

A nested type cannot be declared with the same name as its containing class.

Example:

```
struct S {
    typedef int S;      // Error!
};
```

285 *enumerator '%N' cannot be declared in this class*

An enumerator cannot be declared with the same name as its containing class.

Example:

```
struct S {
    enum E {
        S,      // Error!
        T
    };
};
```

286 *static member '%N' cannot be declared in this class*

A static member cannot be declared with the same name as its containing class.

Example:

```
struct S {
    static int S;       // Error!
};
```

287 *constructor cannot have a return type*

A constructor cannot have a return type (even ***void***). The constructor is a special member function that is not required to be identical in form to all other member functions. This allows different implementations to have different uses for any return values.

Example:

```
class C {
public:
    C& C( int );
};
```

288 *constructor cannot be a static member*

A constructor is a special member function that takes raw storage and changes it into an instance of a class. In order to do this, it must have access to storage for the instance of the class so it is restricted to being a non-static member function.

Example:

```
class C {
public:
    static C( int );
};
```

289 *invalid copy constructor argument list (causes infinite recursion)*

A copy constructor's first argument must be a reference argument. Furthermore, any default arguments must also be reference arguments. Without the reference, a copy constructor would require a copy constructor to execute in order to prepare its arguments. Unfortunately, this would be calling itself since it is the copy constructor.

Example:

```
struct S {
    S( S const & );   // copy constructor
};
```

290 *constructor cannot be declared 'const' or 'volatile'*

A constructor must be able to operate on all instances of classes regardless of whether they are ***const*** or ***volatile.***

Example:

```
class C {
public:
    C( int ) const;
    C( float ) volatile;
};
```

291 *constructor cannot be 'virtual'*

Virtual functions cannot be called for an object before it is constructed. For this reason, a virtual constructor is not allowed in the C++ language. Techniques for simulating a virtual constructor are known, one such technique is described in the ARM p.263.

Example:

```
class C {
public:
    virtual C( int );
};
```

292 *types do not match in simple type destructor*

A simple type destructor is available for "destructing" simple types. The destructor has no effect. Both of the types must be identical, for the destructor to have meaning.

Example:

```
void foo( int *p )
{
    p->int::~double();
}
```

293 *overloaded operator is ambiguous for operands used*

The Watcom C++ compiler performs exhaustive analysis using formalized techniques in order to decide what implicit conversions should be applied for overloading operators. Because of this, Watcom C++ detects ambiguities that may escape other C++ compilers. The most common ambiguity that Watcom C++ detects involves classes having constructors with single arguments and a user-defined conversion.

Example:

```
struct S {
    S(int);
    operator int();
    int a;
};

int fn( int b, int i, S s )
{
    //    i    : s.operator int()
    // OR S(i) : s
    return b ? i : s;
}
```

In the above example, "i" and "s" must be brought to a common type. Unfortunately, there are two common types so the compiler cannot decide which one it should choose, hence an ambiguity.

294 *feature not implemented*

The compiler does not support the indicated feature.

295 *invalid friend declaration*

This message indicates that the compiler found extra declaration specifiers like ***auto, float,*** or ***const*** in the friend declaration.

Example:

```
class C
{
    friend float;
};
```

296 *friend declarations may only be declared in a class*

This message indicates that a ***friend*** declaration was found outside a class scope (i.e., a class definition). Friends are only meaningful for class types.

Example:

```
extern void foo();
friend void foo();
```

297 *class friend declaration needs 'class' or 'struct' keyword*

The C++ language has evolved to require that all friend class declarations be of the form "class S" or "struct S". The Watcom C++ compiler accepts the older syntax with a warning but rejects the syntax in pure ISO/ANSI C++ mode.

Example:

```
struct S;
struct T {
    friend S;   // should be "friend class S;"
};
```

298 *class friend declarations cannot contain a class definition*

A class friend declaration cannot define a new class. This is a restriction required in the C++ language.

Example:

```
struct S {
    friend struct X {
        int f;
    };
};
```

299 *'%T' has already been declared as a friend*

The class in the message has already been declared as a friend. Remove the extra friend declaration.

Example:

```
class S;
class T {
    friend class S;
    int tv;
    friend class S;
};
```

300 *function '%S' has already been declared as a friend*

The function in the message has already been declared as a friend. Remove the extra friend declaration.

Example:

```
extern void foo();
class T {
    friend void foo();
    int tv;
    friend void foo();
};
```

301 *'friend', 'virtual' or 'inline' modifiers are not part of a function's type*

This message indicates that the modifiers may be incorrectly placed in the declaration. If the declaration is intended, it cannot be accepted because the modifiers can only be applied to functions that have code associated with them.

Example:

```
typedef friend (*PF)( void );
```

302 *cannot assign right expression to element on left*

This message indicates that the assignment cannot be performed. It usually arises in assignments of a class type to an arithmetic type.

Example:

```
struct S
{   int sv;
};
S s;
int foo()
{
    int k;
    k = s;
    return k;
}
```

303 *constructor is ambiguous for operands used*

The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

S x = S(1.0);
```

304 *'class %s' has not been defined*

The name before a '::' scope resolution operator must be defined unless a member pointer is being declared.

Example:

```
struct S;

int S::* p;     // OK
int S::a = 1;   // Error!
```

305 *all bit-fields in a union must be named*

This is a restriction in the C++ language. The same effect can be achieved with a named bitfield.

Example:

```
union u
{   unsigned bit1 :10;
    unsigned :6;
};
```

306 *cannot convert expression to type of cast*

The cast is trying to convert an expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended cast.

Example:

```
struct T {
};

void fn()
{
    T y = (T) 0;
}
```

307 *conversion ambiguity: [expression] to [cast type]*

The cast caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

void fn()
{
    S x = (S) 1.0;
}
```

308 *an anonymous class without a declarator is useless*

There is no way to reference the type in this kind of declaration. A name must be provided for either the class or a variable using the class as its type.

Example:

```
struct {
    int a;
    int b;
};
```

309 *global anonymous union must be declared 'static'*

This is a restriction in the C++ language. Since there is no unique name for the anonymous union, it is difficult for C++ translators to provide a correct implementation of external linkage anonymous unions.

Example:

```
static union {
    int a;
    int b;
};
```

310 *anonymous struct/union cannot have storage class in this context*

Anonymous unions (or structs) declared in class scopes cannot be ***static.*** Any other storage class is also disallowed.

Example:

```
struct S {
    static union {
        int iv;
        unsigned us;
    };
};
```

311 *union contains a 'protected' member*

A union cannot have a ***protected*** member because a union cannot be a base class.

Example:

```
static union {
    int iv;
protected:
    unsigned sv;
} u;
```

312 *anonymous struct/union contains a private member '%S'*

An anonymous union (or struct) cannot have member functions or friends so it cannot have ***private*** members since no code could access them.

Example:

```
static union {
    int iv;
private:
    unsigned sv;
};
```

313 *anonymous struct/union contains a function member '%S'*

An anonymous union (or struct) cannot have any function members. This is a restriction in the C++ language.

Example:

```
static union {
    int iv;
    void foo();     // error
    unsigned sv;
};
```

314 *anonymous struct/union contains a typedef member '%S'*

An anonymous union (or struct) cannot have any nested types. This is a restriction in the C++ language.

Example:

```
static union {
    int iv;
    unsigned sv;
    typedef float F;
    F fv;
};
```

315 *anonymous struct/union contains an enumeration member '%S'*

An anonymous union (or struct) cannot have any enumeration members. This is a restriction in the C++ language.

Example:

```
static union {
    int iv;
    enum choice { good, bad, indifferent };
    choice c;
    unsigned sv;
};
```

316 *anonymous struct/union member '%s' is not distinct in enclosing scope*

Since an anonymous union (or struct) provides its member names to the enclosing scope, the names must not collide with other names in the enclosing scope.

Example:

```
int iv;
unsigned sv;
static union {
    int iv;
    unsigned sv;
};
```

317 *unions cannot have members with destructors*

A union should only be used to organize memory in C++. Allowing union members to have destructors would mean that the same piece of memory could be destructed twice.

Example:

```
struct S {
    int sv1, sv2, sv3;
};
struct T {
    ~T();
};
static union
{
    S su;
    T tu;
};
```

318 *unions cannot have members with user-defined assignment operators*

A union should only be used to organize memory in C++. Allowing union members to have assignment operators would mean that the same piece of memory could be assigned twice.

Example:

```
struct S {
    int sv1, sv2, sv3;
};
struct T {
    int tv;
    operator = ( int );
    operator = ( float );
};
static union
{
    S su;
    T tu;
} u;
```

319 *anonymous struct/union cannot have any friends*

An anonymous union (or struct) cannot have any friends. This is a restriction in the C++ language.

Example:

```
struct S {
    int sv1, sv2, sv3;
};
static union {
    S su1;
    S su2;
    friend class S;
};
```

320 *specific versions of template classes can only be defined in file scope*

Currently, specific versions of class templates can only be declared at file scope. This simple restriction was chosen in favour of more freedom with possibly subtle restrictions.

Example:

```
template <class G> class S {
    G x;
};

struct Q {
    struct S<int> {
        int x;
    };
};

void foo()
{
    struct S<double> {
        double x;
    };
}
```

321 *anonymous union in a function may only be 'static' or 'auto'*

The current C++ language definition only allows ***auto*** anonymous unions. The Watcom C++ compiler allows ***static*** anonymous unions. Any other storage class is not allowed.

322 *static data members are not allowed in a local class*

Static data members are not allowed in a local class because there is no way to define the static member in file scope.

Example:

```
int foo()
{
    struct local {
        static int s;
    };

    local lv;

    lv.s = 3;
    return lv.s;
}
```

323 *conversion ambiguity: [return value] to [return type of function]*

The cast caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

S fn()
{
    return 1.0;
}
```

324 *conversion of return value is impossible*

The return is trying to convert an expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended return type.

Example:

```
struct T {
};

T fn()
{
    return 0;
}
```

325 *function cannot return a pointer based on __self*

A function cannot return a pointer that is based on ***__self.***

Example:

```
void __based(__self) *fn( unsigned );
```

326 *defining '%S' is not possible because its type has unknown size*

In order to define a variable, the size must be known so that the correct amount of storage can be reserved.

Example:

```
class S;
S sv;
```

327 *typedef cannot be initialized*

Initializing a ***typedef*** is meaningless in the C++ language.

Example:

```
typedef int INT = 15;
```

328 *storage class of '%S' conflicts with previous declaration*

The symbol declaration conflicts with a previous declaration with regard to storage class. A symbol cannot be both ***static*** and ***extern.***

329 *modifiers of '%S' conflict with previous declaration*

The symbol declaration conflicts with a previous declaration with regard to modifiers. Correct the program by using the same modifiers for both declarations.

330 *function cannot be initialized*

A function cannot be initialized with an initializer syntax intended for variables. A function body is the only way to provide a definition for a function.

331 *access permission of nested class '%T' conflicts with previous declaration*

Example:

```
struct S {
    struct N;   // public
private:
    struct N {  // private
    };
};
```

332 **** FATAL *** internal error in front end*

If this message appears, please report the problem directly to Watcom.

333 *cannot convert argument to type specified in function prototype*

It is impossible to convert the indicated argument in the function.

Example:

```
extern int foo( int& );

extern int m;
extern int n;

int k = foo( m + n );
```

In the example, the value of "m+n" cannot be converted to a reference (it could be converted to a constant reference), as shown in the following example.

Example:

```
extern int foo( const int& );

extern int m;
extern int n;

int k = foo( m + n );
```

334 *conversion ambiguity: [argument] to [argument type in prototype]*

An argument in the function call could not be converted since there is more than one constructor or user-defined conversion which could be used to convert the argument.

Example:

```
struct S;

struct T
{
    T( S& );
};

struct S
{
    operator T();
};

S s;
extern int foo( T );
int k = foo( s );   // ambiguous
```

In the example, the argument "s" could be converted by both the constructor in class "T" and by the user-conversion in class "S".

335 *cannot be based on based pointer '%S'*

A based pointer cannot be based on another based pointer.

Example:

```
__segment s;
void __based(s) *p;
void __based(p) *q;
```

336 *declaration specifiers are required to declare '%N'*

The compiler has detected that the name does not represent a function. Only function declarations can leave out declaration specifiers. This error also shows up when a typedef name declaration is missing.

Example:

```
x;
typedef int;
```

337 *static function declared in block scope*

The C++ language does not allow static functions to be declared in block scope. This error can be triggered when the intent is to define a ***static*** variable. Due to the complexities of parsing C++, statements that appear to be variable definitions may actually parse as function prototypes. A work-around for this problem is contained in the example.

Example:

```
struct C {
};
struct S {
    S( C );
};
void foo()
{
    static S a( C() );  // function prototype!
    static S b( (C()) );// variable definition
}
```

338 *cannot define a __based reference*

A C++ reference cannot be based on anything. Based modifiers can only be used with pointers.

Example:

```
__segment s;
void fn( int __based(s) & x );
```

339 *conversion ambiguity: conversion to common pointer type*

A conversion to a common base class of two different pointers has been attempted. The pointer conversion could not be performed because the destination type points to an ambiguous base class of one of the source types.

340 *cannot construct object from argument(s)*

There is not an appropriate constructor for the set of arguments provided.

341 *number of arguments for function '%S' is incorrect*

The number of arguments in the function call does not match the number declared for the indicated non-overloaded function.

Example:

```
extern int foo( int, int );
int k = foo( 1, 2, 3 );
```

In the example, the function was declared to have two arguments. Three arguments were used in the call.

342 *private base class accessed to convert cast expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

extern Der *pd;
Priv *pp = (Priv*)pd;
```

343 *private base class accessed to convert return expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

Priv *foo( Der *p )
{
    return p;
}
```

344 *cannot subtract pointers to different objects*

Pointer subtraction can be performed only for objects of the same type.

Example:

```
#include <stddef.h>
ptrdiff_t diff( float *fp, int *ip )
{
    return fp - ip;
}
```

In the example, a diagnostic results from the attempt to subtract a pointer to an ***int*** object from a pointer to a ***float*** object.

345 *private base class accessed to convert to common pointer type*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Priv
{
    int p;
};
struct Der : private Priv
{
    int d;
};

int foo( Der *pd, Priv *pp )
{
    return pd == pp;
}
```

346 *protected base class accessed to convert cast expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

extern Der *pd;
Prot *pp = (Prot*)pd;
```

347 *protected base class accessed to convert return expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

Prot *foo( Der *p )
{
    return p;
}
```

348 *cannot define a member pointer with a memory model modifier*

A member pointer describes how to access a field from a class. Because of this a member pointer must be independent of any memory model considerations.

Example:

```
struct S;

int near S::*mp;
```

349 *protected base class accessed to convert to common pointer type*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
struct Prot
{
    int p;
};
struct Der : protected Prot
{
    int d;
};

int foo( Der *pd, Prot *pp )
{
    return pd == pp;
}
```

350 *non-type parameter supplied for a type argument*

A non-type parameter (e.g., an address or a constant expression) has been supplied for a template type argument. A type should be used instead.

351 *type parameter supplied for a non-type argument*

A type parameter (e.g., ***int***) has been supplied for a template non-type argument. An address or a constant expression should be used instead.

352 *cannot access enclosing function's auto variable '%S'*

A local class member function cannot access its enclosing function's automatic variables.

Example:

```
void goop( void )
{
    int a;
    struct S
    {
        int foo( int c, int b )
            {
                return b + c + a;
            };
    };
}
```

353 *cannot initialize pointer to non-constant with a pointer to constant*

A pointer to a non-constant type cannot be initialized with a pointer to a constant type because this would allow constant data to be modified via the non-constant pointer to it.

Example:

```
extern const int *pic;
extern int *pi = pic;
```

354 *pointer expression is always >= 0*

The indicated pointer expression will always be true because the pointer value is always treated as an unsigned quantity, which will be greater or equal to zero.

Example:

```
extern char *p;
unsigned k = ( 0 <= p );    // always 1
```

355 *pointer expression is never < 0*

The indicated pointer expression will always be false because the pointer value is always treated as an unsigned quantity, which will be greater or equal zero.

Example:

```
extern char *p;
unsigned k = ( 0 >= p );   // always 0
```

356 *type cannot be used in this context*

This message is issued when a type name is being used in a context where a non-type name should be used.

Example:

```
struct S {
    typedef int T;
};

void fn( S *p )
{
    p->T = 1;
}
```

357 *virtual function may only be declared in a class*

Virtual functions can only be declared inside of a class. This error may be a result of forgetting the "C::" qualification of a virtual function’s name.

Example:

```
virtual void foo();
struct S
{
    int f;
    virtual void bar();
};
virtual void bar()
{
    f = 9;
}
```

358 *'%T' referenced as a union*

A class type defined as a ***class*** or ***struct*** has been referenced as a ***union*** (i.e., union S).

Example:

```
struct S
{
    int s1, s2;
};
union S var;
```

359 *'union %T' referenced as a class*

A class type defined as a ***union*** has been referenced as a ***struct*** or a ***class*** (i.e., class S).

Example:

```
union S
{
    int s1, s2;
};
struct S var;
```

360 *typedef '%N' defined without an explicit type*

The typedef declaration was found to not have an explicit type in the declaration. If ***int*** is the desired type, use an explicit ***int*** keyword to specify the type.

Example:

```
typedef T;
```

361 *member function was not defined in its class*

Member functions of local classes must be defined in their class if they will be defined at all. This is a result of the C++ language not allowing nested function definitions.

Example:

```
void fn()
{
    struct S {
        int bar();
    };
}
```

362 *local class can only have its containing function as a friend*

A local class can only be referenced from within its containing function. It is impossible to define an external function that can reference the type of the local class.

Example:

```
extern void ext();
void foo()
{
    class S
    {
        int s;
    public:
        friend void ext();
        int q;
    };
}
```

363 *local class cannot have '%S' as a friend*

The only classes that a local class can have as a friend are classes within its own containing scope.

Example:

```
struct ext
{
    goop();
};
void foo()
{
    class S
    {
        int s;
    public:
        friend class ext;
        int q;
    };
}
```

364 *adjacent >=, <=, >, < operators*

This message is warning about the possibility that the code may not do what was intended. An expression like "a > b > c" evaluates one relational operator to a 1 or a 0 and then compares it against the other variable.

Example:

```
extern int a;
extern int b;
extern int c;
int k = a > b > c;
```

365 *cannot access enclosing function's argument '%S'*

A local class member function cannot access its enclosing function's arguments.

Example:

```
void goop( int d )
{
    struct S
    {
        int foo( int c, int b )
            {
                return b + c + d;
            };
    };
}
```

366 *support for switch '%s' is not implemented*

Actions for the indicated switch have not been implemented. The switch is supported for compatibility with the Watcom C compiler.

367 *conditional expression in if statement is always true*

The compiler has detected that the expression will always be true. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially true expressions.

Example:

```
#define TEST 143
int foo( int a, int b )
{
    if( TEST ) return a;
    return b;
}
```

368 *conditional expression in if statement is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

Example:

```
#define TEST 14-14
int foo( int a, int b )
{
    if( TEST ) return a;
    return b;
}
```

369 *selection expression in switch statement is a constant value*

The expression in the ***switch*** statement is a constant. This means that only one case label will be executed. If this is not the expected behaviour, check the switch expression.

Example:

```
#define TEST 0
int foo( int a, int b )
{
    switch ( TEST ) {
      case 0:
        return a;
      default:
        return b;
    }
}
```

370 *constructor is required for a class with a const member*

If a class has a constant member, a constructor is required in order to initialize it.

Example:

```
struct S
{
    const int s;
    int i;
};
```

371 *constructor is required for a class with a reference member*

If a class has a reference member, a constructor is required in order to initialize it.

Example:

```
struct S
{
    int& r;
    int i;
};
```

372 *inline member friend function '%S' is not allowed*

A friend that is a member function of another class cannot be defined. Inline friend rules are currently in flux so it is best to avoid inline friends.

373 *invalid modifier for auto variable*

An automatic variable cannot have a memory model adjustment because they are always located on the stack (or in a register). There are also other types of modifiers that are not allowed for auto variables such as thread-specific data modifiers.

Example:

```
int fn( int far x )
{
    int far y = x + 1;
    return y;
}
```

374 *object (or object pointer) required to access non-static data member*

A reference to a member in a class has occurred. The member is non-static so in order to access it, an object of the class is required.

Example:

```
struct S {
    int m;
    static void fn()
    {
        m = 1;  // Error!
    }
};
```

375 *user-defined conversion has not been declared*

The named user-defined conversion has not been declared in the class of any of its base classes.

Example:

```
struct S {
    operator int();
    int a;
};

double fn( S *p )
{
    return p->operator double();
}
```

376 *virtual function must be a non-static member function*

A member function cannot be both a ***static*** function and a ***virtual*** function. A static member function does not have a ***this*** argument whereas a ***virtual*** function must have a ***this*** argument so that the virtual function table can be accessed in order to call it.

Example:

```
struct S
{
    static virtual int foo();   // error
    virtual int bar();          // ok
    static int stat();          // ok
};
```

377 *protected base class accessed to convert argument expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

Example:

```
class C
{
protected:
    C( int );
public:
    int c;
};

int cfun( C );

int i = cfun( 14 );
```

The last line is erroneous since the constructor is protected.

378 *private base class accessed to convert argument expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

Example:

```
class C
{
    C( int );
public:
    int c;
};

int cfun( C );

int i = cfun( 14 );
```

The last line is erroneous since the constructor is private.

379 *'delete' expression will invoke a non-virtual destructor*

In C++, it is possible to assign a base class pointer the value of a derived class pointer so that code that makes use of base class virtual functions can be used. A problem that occurs is that a ***delete*** has to know the correct size of the type in some instances (i.e., when a two argument version of ***operator delete*** is defined for a class). This problem is solved by requiring that a destructor be defined as ***virtual*** if polymorphic deletes must work. The ***delete*** expression will virtually call the correct destructor, which knows the correct size of the complete object. This message informs you that the class you are deleting has virtual functions but it has a non-virtual destructor. This means that the delete will not work correctly in all circumstances.

Example:

```
#include <stddef.h>

struct B {
    int b;
    void operator delete( void *, size_t );
    virtual void fn();
    ~B();
};
struct D : B {
    int d;
    void operator delete( void *, size_t );
    virtual void fn();
    ~D();
};

void dfn( B *p )
{
    delete p;   // could be a pointer to D!
}
```

380 *'offsetof' is not allowed for a function*

A member function does not have an offset like simple data members. If this is required, use a member pointer.

Example:

```
#include <stddef.h>

struct S
{
    int fun();
};

int s = offsetof( S, fun );
```

381 *'offsetof' is not allowed for an enumeration*

An enumeration does not have an offset like simple data members.

Example:

```
#include <stddef.h>

struct S
{
    enum SE { S1, S2, S3, S4 };
    SE var;
};

int s = offsetof( S, SE );
```

382 *could not initialize for code generation*

The source code has been parsed and fully analysed when this error is emitted. The compiler attempted to start generating object code but due to some problem (e.g., out of memory, no file handles) could not initialize itself. Try changing the compilation environment to eliminate this error.

383 *'offsetof' is not allowed for an undefined type*

The class type used in ***offsetof*** must be completely defined, otherwise data member offsets will not be known.

Example:

```
#include <stddef.h>

struct S {
    int a;
    int b;
    int c[ offsetof( S, b ) ];
};
```

384 *attempt to override virtual function '%S' with a different return type*

A function cannot be overloaded with identical argument types and a different return type. This is due to the fact that the C++ language does not consider the function's return type when overloading. The exception to this rule in the C++ language involves restricted changes in the return type of virtual functions. The derived virtual function's return type can be derived from the return type of the base virtual function.

Example:

```
struct B {
    virtual B *fn();
};
struct D : B {
    virtual D *fn();
};
```

385 *attempt to overload function '%S' with a different return type*

A function cannot be overloaded with identical argument types and a different return type. This is due to the fact that the C++ language does not consider the function's return type when overloading.

Example:

```
int foo( char );
unsigned foo( char );
```

386 *attempt to use pointer to undefined class*

An attempt was made to indirect or increment a pointer to an undefined class. Since the class is undefined, the size is not known so the compiler cannot compile the expression properly.

Example:

```
class C;
extern C* pc1;
C* pc2 = ++pc1;          // C not defined

int foo( C*p )
{
    return p->x;         // C not defined
}
```

387 *expression is useful only for its side effects*

The indicated expression is not meaningful. The expression, however, does contain one or more side effects.

Example:

```
extern int* i;
void func()
{
    *(i++);
}
```

In the example, the expression is a reference to an integer which is meaningless in itself. The incrementation of the pointer in the expression is a side effect.

388 *integral constant will be truncated during assignment or initialization*

This message indicates that the compiler knows that a constant value will not be preserved after the assignment. If this is acceptable, cast the constant value to the appropriate type in the assignment.

Example:

```
unsigned char c = 567;
```

389 *integral value may be truncated during assignment or initialization*

This message indicates that the compiler knows that all values will not be preserved after the assignment. If this is acceptable, cast the value to the appropriate type in the assignment.

Example:

```
extern unsigned s;
unsigned char c = s;
```

390 *cannot generate default constructor to initialize '%T' since constructors were declared*

A default constructor will not be generated by the compiler if there are already constructors declared. Try using default arguments to change one of the constructors to a default constructor or define a default constructor explicitly.

Example:

```
class C {
    C( const C& );
public :
    int c;
};
C cv;
```

391 *assignment found in boolean expression*

This is a construct that can lead to errors if it was intended to be an equality (using "==") test.

Example:

```
int foo( int a, int b )
{
    if( a = b ) {
        return b;
    }
    return a;           // always return 1 ?
}
```

392 *'%F' defined %L*

This informational message indicates where the symbol in question was defined. The message is displayed following an error or warning diagnostic for the symbol in question.

Example:

```
static int a = 9;
int b = 89;
```

The variable 'a' is not referenced in the preceding example and so will cause a warning to be generated. Following the warning, the informational message indicates the line at which 'a' was declared.

393 *included from %s(%u)*

This informational message indicates the line number of the file including the file in which an error or warning was diagnosed. A number of such messages will allow you to trace back through the ***#include*** directives which are currently being processed.

394 *reference object must be initialized*

A reference cannot be set except through initialization. Also references cannot be 0 so they must always be initialized.

Example:

```
int & ref;
```

395 *option requires an identifier*

The specified option is not recognized by the compiler since there was no identifier after it (i.e., "-nt=module").

396 *'main' cannot be overloaded*

There can only be one entry point for a C++ program. The "main" function cannot be overloaded.

Example:

```
int main();
int main( int );
```

397 *'new' expression cannot allocate a 'void'*

Since the ***void*** type has no size and there are no values of ***void*** type, one cannot allocate an instance of ***void.***

Example:

```
void *p = new void;
```

398 *'new' expression cannot allocate a function*

A function type cannot be allocated since there is no meaningful size that can be used. The ***new*** expression can allocate a pointer to a function.

Example:

```
typedef int tdfun( int );
tdfun *tdv = new tdfun;
```

399 *'new' expression allocates a 'const' or 'volatile' object*

The pool of raw memory cannot be guaranteed to support ***const*** or ***volatile*** semantics. Usually ***const*** and ***volatile*** are used for statically allocated objects.

Example:

```
typedef const int con_int;
con_int* p = new con_int;
```

400 *cannot convert right expression for initialization*

The initialization is trying to convert an argument expression to a completely unrelated type. There is no way the compiler can provide any meaning for the intended conversion.

Example:

```
struct T {
};

T x = 0;
```

401 *conversion ambiguity: [initialization expression] to [type of object]*

The initialization caused a constructor overload to occur. The operands provided for the constructor did not select a unique constructor.

Example:

```
struct S {
    S(int);
    S(char);
};

S x = 1.0;
```

402 *class template '%S' has already been declared as a friend*

The class template in the message has already been declared as a friend. Remove the extra friend declaration.

Example:

```
template <class T>
    class S;

class X {
    friend class S;
    int f;
    friend class S;
};
```

403 *private base class accessed to convert initialization expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

404 *protected base class accessed to convert initialization expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

405 *cannot return a pointer or reference to a constant object*

A pointer or reference to a constant object cannot be returned.

Example:

```
int *foo( const int *p )
{
    return p;
}
```

406 *cannot pass a pointer or reference to a constant object*

A pointer or reference to a constant object could not be passed as an argument.

Example:

```
int *bar( int * );
int *foo( const int *p )
{
    return bar( p );
}
```

407 *class templates must be named*

There is no syntax in the C++ language to reference an unnamed class template.

Example:

```
template <class T>
    class {
    };
```

408 *function templates can only name functions*

Variables cannot be overloaded in C++ so it is not possible to have many different instances of a variable with different types.

Example:

```
template <class T>
    T x[1];
```

409 *template argument '%S' is not used in the function argument list*

This restriction ensures that function templates can be bound to types during overload resolution. Functions currently can only be overloaded based on argument types.

Example:

```
template <class T>
    int foo( int * );
template <class T>
    T bar( int * );
```

410 *destructor cannot be declared 'const' or 'volatile'*

A destructor must be able to operate on all instances of classes regardless of whether they are ***const*** or ***volatile.***

411 *static member function cannot be declared 'const' or 'volatile'*

A static member function does not have an implicit ***this*** argument so the ***const*** and ***volatile*** function qualifiers cannot be used.

412 *only member functions can be declared 'const' or 'volatile'*

A non-member function does not have an implicit ***this*** argument so the ***const*** and ***volatile*** function qualifiers cannot be used.

413 *'const' or 'volatile' modifiers are not part of a function's type*

The ***const*** and ***volatile*** qualifiers for a function cannot be used in typedefs or pointers to functions. The trailing qualifiers are used to change the type of the implicit ***this*** argument so that member functions that do not modify the object can be declared accurately.

Example:

```
// const is illegal
typedef void (*baddcl)() const;

struct S {
    void fun() const;
    int a;
};

// "this" has type "S const *"
void S::fun() const
{
    this->a = 1;    // Error!
}
```

414 *type cannot be defined in an argument*

A new type cannot be defined in an argument because the type will only be visible within the function. This amounts to defining a function that can never be called because C++ uses name equivalence for type checking.

Example:

```
extern foo( struct S { int s; } );
```

415 *type cannot be defined in return type*

This is a restriction in the current C++ language. A function prototype should only use previously declared types in order to guarantee that it can be called from other functions. The restriction is required for templates because the compiler would have to wait until the end of a class definition before it could decide whether a class template or function template is being defined.

Example:

```
template <class T>
    class C {
        T value;
    } fn( T x ) {
        C y;

        y.x = 0;
        return y;
    };
```

A common problem that results in this error is to forget to terminate a class or enum definition with a semicolon.

Example:

```
struct S {
    int x,y;
    S( int, int );
} // missing semicolon ';'

S::S( int x, int y ) : x(x), y(y) {
}
```

416 *data members cannot be initialized inside a class definition*

This message appears when an initialization is attempted inside of a class definition. In the case of static data members, initialization must be done outside the class definition. Ordinary data members can be initialized in a constructor.

Example:

```
struct S {
    static const int size = 1;
};
```

417 *only virtual functions may be declared pure*

The C++ language requires that all pure functions be declared virtual. A pure function establishes an interface that must consist of virtual functions because the functions are required to be defined in the derived class.

Example:

```
struct S {
    void foo() = 0;
};
```

418 *destructor is not declared in its proper class*

The destructor name is not declared in its own class or qualified by its own class. This is required in the C++ language.

419 *cannot call non-const function for a constant object*

A function that does not promise to not modify an object cannot be called for a constant object. A function can declare its intention to not modify an object by using the ***const*** qualifier.

Example:

```
struct S {
    void fn();
};

void cfn( const S *p )
{
    p->fn();    // Error!
}
```

420 *memory initializer list may only appear in a constructor definition*

A memory initializer list should be declared along with the body of the constructor function.

421 *cannot initialize member '%N' twice*

A member cannot be initialized twice in a member initialization list.

422 *cannot initialize base class '%T' twice*

A base class cannot be constructed twice in a member initialization list.

423 *'%T' is not a direct base class*

A base class initializer in a member initialization list must either be a direct base class or a virtual base class.

424 *'%N' cannot be initialized because it is not a member*

The name used in the member initialization list does not name a member in the class.

425 *'%N' cannot be initialized because it is a member function*

The name used in the member initialization list does not name a non-static data member in the class.

426 *'%N' cannot be initialized because it is a static member*

The name used in the member initialization list does not name a non-static data member in the class.

427 *'%N' has not been declared as a member*

This message indicates that the member does not exist in the qualified class. This usually occurs in the context of access declarations.

428 *const/reference member '%S' must have an initializer*

The ***const*** or reference member does not have an initializer so the constructor is not completely defined. The member initialization list is the only way to initialize these types of members.

429 *abstract class '%T' cannot be used as an argument type*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as an argument type.

430 *abstract class '%T' cannot be used as a function return type*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as a return type.

431 *defining '%S' is not possible because '%T' is an abstract class*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as either a member or a variable.

432 *cannot convert to an abstract class '%T'*

An abstract class can only exist as a base class of another class. The C++ language does not allow an abstract class to be used as the destination type in a conversion.

433 *mangled name for '%S' has been truncated*

The name used in the object file that encodes the name and full type of the symbol is often called a mangled name. The warning indicates that the mangled name had to be truncated due to limitations in the object file format.

434 *cannot convert to a type of unknown size*

A completely unknown type cannot be used in a conversion because its size is not known. The behaviour of the conversion would be undefined also.

435 *cannot convert a type of unknown size*

A completely unknown type cannot be used in a conversion because its size is not known. The behaviour of the conversion would be undefined also.

436 *cannot construct an abstract class*

An instance of an abstract class cannot be created because an abstract class can only be used as a base class.

437 *cannot construct an undefined class*

An instance of an undefined class cannot be created because the size is not known.

438 *string literal concatenated during array initialization*

This message indicates that a missing comma (',') could have made a quiet change in the program. Otherwise, ignore this message.

439 *maximum size of segment '%s' has been exceeded for '%S'*

The indicated symbol has grown in size to a point where it has caused the segment it is defined inside of to be exhausted.

440 *maximum data item size has been exceeded for '%S'*

A non-huge data item is larger than 64k bytes in size. This message only occurs during 16-bit compilation of C++ code.

441 *function attribute has been repeated*

A function attribute (like the ***__export*** attribute) has been repeated. Remove the extra attribute to correct the declaration.

442 *modifier has been repeated*

A modifier (like the ***far*** modifier) has been repeated. Remove the extra modifier to correct the declaration.

443 *illegal combination of memory model modifiers*

Memory model modifiers must be used individually because they cannot be combined meaningfully.

444 *argument name '%N' has already been used*

The indicated argument name has already been used in the same argument list. This is not allowed in the C++ language.

445 *function definition for '%S' must be declared with an explicit argument list*

A function cannot be defined with a typedef. The argument list must be explicit.

446 *user-defined conversion cannot convert to its own class or base class*

A user-defined conversion cannot be declared as a conversion either to its own class or to a base class of itself.

Example:

```
struct B {
};
struct D : private B {
    operator B();
};
```

447 *user-defined conversion cannot convert to 'void'*

A user-defined conversion cannot be declared as a conversion to ***void.***

Example:

```
struct S {
    operator void();
};
```

448 *expecting identifier*

An identifier was expected during processing.

449 *symbol '%S' does not have a segment associated with it*

A pointer cannot be based on a member because it has no segment associated with it. A member describes a layout of storage that can occur in any segment.

450 *symbol '%S' must have integral or pointer type*

If a symbol is based on another symbol, it must be integral or a pointer type. An integral type indicates the segment value that will be used. A pointer type means that all accesses will be added to the pointer value to construct a full pointer.

451 *symbol '%S' cannot be accessed in all contexts*

The symbol that the pointer is based on is in another class so it cannot be accessed in all contexts that the based pointer can be accessed.

452 *cannot convert class expression to be copied*

A convert class expression could not be copied.

453 *conversion ambiguity: multiple copy constructors*

More than one constructor could be used to copy a class object.

454 *function template '%S' already has a definition*

The function template has already been defined with a function body. A function template cannot be defined twice even if the function body is identical.

Example:

```
template <class T>
    void f( T *p )
    {
    }
template <class T>
    void f( T *p )
    {
    }
```

455 *function templates cannot have default arguments*

A function template must not have default arguments because there are certain types of default arguments that do not force the function argument to be a specific type.

Example:

```
template <class T>
    void f2( T *p = 0 )
    {
    }
```

456 *'main' cannot be a function template*

This is a restriction in the C++ language because "main" cannot be overloaded. A function template provides the possibility of having more than one "main" function.

457 *'%S' was previously declared as a typedef*

The C++ language only allows function and variable names to coexist with names of classes or enumerations. This is due to the fact that the class and enumeration names can still be referenced in their elaborated form after the non-type name has been declared.

Example:

```
typedef int T;
int T( int )            // error!
{
}

enum E { A, B, C };
void E()
{
    enum E x = A;       // use "enum E"
}

class C { };
void C()
{
    class C x;          // use "class C"
}
```

458 *'%S' was previously declared as a variable/function*

The C++ language only allows function and variable names to coexist with names of classes or enumerations. This is due to the fact that the class and enumeration names can still be referenced in their elaborated form after the non-type name has been declared.

Example:

```
int T( int )
{
}
typedef int T;          // error!

void E()
{
}
enum E { A, B, C };

enum E x = A;           // use "enum E"

void C()
{
}
class C { };

class C x;              // use "class C"
```

459 *private base class accessed to convert assignment expression*

A conversion involving the inheritance hierarchy required access to a private base class. The access check did not succeed so the conversion is not allowed.

460 *protected base class accessed to convert assignment expression*

A conversion involving the inheritance hierarchy required access to a protected base class. The access check did not succeed so the conversion is not allowed.

461 *maximum size of DGROUP has been exceeded for '%S' in segment '%s'*

The indicated symbol's size has caused the DGROUP contribution of this module to exceed 64k. Changing memory models or declaring some data as ***far*** data are two ways of fixing this problem.

462 *type of return value is not the enumeration type of function*

The return value does not have the proper enumeration type. Keep in mind that integral values are not automatically converted to enum types like the C language.

463 *linkage must be first in a declaration; probable cause: missing ';'*

This message usually indicates a missing semicolon (';'). The linkage specification must be the first part of a declaration if it is used.

464 *'main' cannot be a static function*

This is a restriction in the C++ language because "main" must have external linkage.

465 *'main' cannot be an inline function*

This is a restriction in the C++ language because "main" must have external linkage.

466 *'main' cannot be referenced*

This is a restriction in the C++ language to prevent implementations from having to work around multiple invocations of "main". This can occur if an implementation has to generate special code in "main" to construct all of the statically allocated classes.

467 *cannot call a non-volatile function for a volatile object*

A function that does not promise to not modify an object using ***volatile*** semantics cannot be called for a volatile object. A function can declare its intention to modify an object only through ***volatile*** semantics by using the ***volatile*** qualifier.

Example:

```
struct S {
    void fn();
};

void cfn( volatile S *p )
{
    p->fn();    // Error!
}
```

468 *cannot convert pointer to constant or volatile objects to 'void*'*

You cannot convert a pointer to constant or volatile objects to 'void*'.

Example:

```
extern const int* pci;
extern void *vp;

int k = ( pci == vp );
```

469 *cannot convert pointer to constant or non-volatile objects to 'volatile void*'*

You cannot convert a pointer to constant or non-volatile objects to 'volatile void*'.

Example:

```
extern const int* pci;
extern volatile void *vp;

int k = ( pci == vp );
```

470 *address of function is too large to be converted to 'void*'*

The address of a function can be converted to 'void*' only when the size of a 'void*' object is large enough to contain the function pointer.

Example:

```
void __far foo();
void __near *v = &foo;
```

471 *address of data object is too large to be converted to 'void*'*

The address of an object can be converted to 'void*' only when the size of a 'void*' object is large enough to contain the pointer.

Example:

```
int __far *ip;
void __near *v = ip;
```

472 *expression with side effect in 'sizeof' discarded*

The indicated expression will be discarded; consequently, any side effects in that expression will not be executed.

Example:

```
int a = 14;
int b = sizeof( a++ );
```

In the example, the variable `a` will still have a value 14 after `b` has been initialized.

473 *function argument(s) do not match those in prototype*

The C++ language requires great precision in specifying arguments for a function. For instance, a pointer to `char` is considered different than a pointer to `unsigned char` regardless of whether `char` is an unsigned quantity. This message occurs when a non-overloaded function is invoked and one or more of the arguments cannot be converted. It also occurs when the number of arguments differs from the number specified in the prototype.

474 *conversion ambiguity: [expression] to [class object]*

The conversion of the expression to a class object is ambiguous.

475 *cannot assign right expression to class object*

The expression on the right cannot be assigned to the indicated class object.

476 *argument count is %d since there is an implicit 'this' argument*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is a member function with a ***this*** argument so it may have one more argument than expected.

477 *argument count is %d since there is no implicit 'this' argument*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is a member function without a ***this*** argument so it may have one less argument than expected.

478 *argument count is %d for a non-member function*

This informational message indicates the number of arguments for the function mentioned in the error message. The function is not a member function but it could be declared as a ***friend*** function.

479 *conversion ambiguity: multiple copy constructors to copy array '%S'*

More than one constructor to copy the indicated array exists.

480 *variable/function has the same name as the class/enum '%S'*

In C++, a class or enum name can coexist with a variable or function of the same name in a scope. This warning is indicating that the current declaration is making use of this feature but the typedef name was declared in another file. This usually means that there are two unrelated uses of the same name.

481 *class/enum has the same name as the function/variable '%S'*

In C++, a class or enum name can coexist with a variable or function of the same name in a scope. This warning is indicating that the current declaration is making use of this feature but the function/variable name was declared in another file. This usually means that there are two unrelated uses of the same name. Furthermore, all references to the class or enum must be elaborated (i.e., use 'class C' instead of 'C') in order for subsequent references to compile properly.

482 *cannot create a default constructor*

A default constructor could not be created, because other constructors were declared for the class in question.

Example:

```
struct X {
    X(X&);
};
struct Y {
    X a[10];
};
Y yvar;
```

In the example, the variable "yvar" causes a default constructor for the class "Y" to be generated. The default constructor for "Y" attempts to call the default constructor for "X" in order to initialize the array "a" in class "Y". The default

constructor for "X" cannot be defined because another constructor has been declared.

483 *attempting to access default constructor for %T*

This informational message indicates that a default constructor was referenced but could not be generated.

484 *cannot align symbol '%S' to segment boundary*

The indicated symbol requires more than one segment of storage and the symbol's components cannot be aligned to the segment boundary.

485 *friend declaration does not specify a class or function*

A class or function must be declared as a friend.

Example:

```
struct T {
    // should be class or function declaration
    friend int;
};
```

486 *cannot take address of overloaded function*

This message indicates that an overloaded function's name was used in a context where a final type could not be found. Because a final type was not specified, the compiler cannot select one function to use in the expression. Initialize a properly-typed temporary with the appropriate function and use the temporary in the expression.

Example:

```
int foo( char );
int foo( unsigned );
extern int (*p)( char );
int k = ( p == &foo );              // fails
```

The first `foo` can be passed as follows:

Example:

```
int foo( char );
int foo( unsigned );
extern int (*p)( char );

// introduce temporary
static int (*temp)( char ) = &foo;

// ok
int k = ( p == temp );
```

487 *cannot use address of overloaded function as a variable argument*

This message indicates that an overloaded function's name was used as a argument for a "..." style function. Because a final function type is not present, the compiler cannot select one function to use in the expression. Initialize a properly-typed temporary with the appropriate function and use the temporary in the call.

Example:

```
int foo( char );
int foo( unsigned );
int ellip_fun( int, ... );
int k = ellip_fun( 14, &foo );      // fails
```

The first `foo` can be passed as follows:

Example:

```
int foo( char );
int foo( unsigned );
int ellip_fun( int, ... );

static int (*temp)( char ) = &foo;  // introduce
temporary

int k = ellip_fun( 14, temp );      // ok
```

488 *'%N' cannot be overloaded*

The indicated function cannot be overloaded. Functions that fall into this category include ***operator delete.***

489 *symbol '%S' has already been initialized*

The indicated symbol has already been initialized. It cannot be initialized twice even if the initialization value is identical.

490 *'delete' expression is a pointer to a function*

A pointer to a function cannot be allocated so it cannot be deleted.

491 *delete of a pointer to const data*

Since deleting a pointer may involve modification of data, it is not always safe to delete a pointer to const data.

Example:

```
struct S { };
void fn( S const *p, S const *q ) {
    delete p;
    delete [] q;
}
```

492 *'delete' expression is not a pointer to data*

A ***delete*** expression can only delete pointers. For example, trying to delete an ***int*** is not allowed in the C++ language.

Example:

```
void fn( int a )
{
    delete a;   // Error!
}
```

493 *template argument is not a constant expression*

The compiler has found an incorrect expression provided as the value for a constant value template argument. The only expressions allowed for scalar template arguments are integral constant expressions.

494 *template argument is not an external linkage symbol*

The compiler has found an incorrect expression provided as the value for a pointer value template argument. The only expressions allowed for pointer template arguments are addresses of symbols. Any symbols must have external linkage or must be static class members.

495 *conversion of const reference to volatile reference*

The constant value can be modified by assigning into the volatile reference. This would allow constant data to be modified quietly.

Example:

```
void fn( const int &rci )
{
    int volatile &r = rci;      // Error!
}
```

496 *conversion of volatile reference to const reference*

The volatile value can be read incorrectly by accessing the const reference. This would allow volatile data to be accessed without correct volatile semantics.

Example:

```
void fn( volatile int &rvi )
{
    int const &r = rvi; // Error!
}
```

497 *conversion of const or volatile reference to plain reference*

The constant value can be modified by assigning into the plain reference. This would allow constant data to be modified quietly. In the case of volatile data, any access to the plain reference will not respect the volatility of the data and thus would be incorrectly accessing the data.

Example:

```
void fn( const int &rci, volatile int &rvi )
{
    int &r1 = rci;      // Error!
    int &r2 = rvi;      // Error!
}
```

498 *syntax error before '%s'; probable cause: incorrectly spelled type name*

The identifier in the error message has not been declared as a type name in any scope at this point in the code. This may be the cause of the syntax error.

499 *object (or object pointer) required to access non-static member function*

A reference to a member function in a class has occurred. The member is non-static so in order to access it, an object of the class is required.

Example:

```
struct S {
    int m();
    static void fn()
    {
        m();    // Error!
    }
};
```

500 *object (or object pointer) cannot be used to access function*

The indicated object (or object pointer) cannot be used to access function.

501 *object (or object pointer) cannot be used to access data*

The indicated object (or object pointer) cannot be used to access data.

502 *cannot access member function in enclosing class*

A member function in enclosing class cannot be accessed.

503 *cannot access data member in enclosing class*

A data member in enclosing class cannot be accessed.

504 *syntax error before type name '%s'*

The identifier in the error message has been declared as a type name at this point in the code. This may be the cause of the syntax error.

505 *implementation restriction: cannot generate thunk from '%S'*

This implementation restriction is due to the use of a shared code generator between Watcom compilers. The virtual ***this*** adjustment thunks are generated as functions linked into the virtual function table. The functions rely on knowing the correct number of arguments to pass on to the overriding virtual function but in the case of ellipsis (...) functions, the number of arguments cannot be known when the thunk function is being generated by the compiler. The target symbol is listed in a diagnostic message. The work around for this problem is to recode the source so that the virtual functions make use of the va_list type found in the stdarg header file.

Example:

```
#include <iostream.h>
#include <stdarg.h>

struct B {
    virtual void fun( char *, ... );
};
struct D : B {
    virtual void fun( char *, ... );
};
void B::fun( char *f, ... )
{
    va_list args;

    va_start( args, f );
    while( *f ) {
        cout << va_arg( args, char ) << endl;
        ++f;
    }
    va_end( args );
}
void D::fun( char *f, ... )
{
    va_list args;

    va_start( args, f );
    while( *f ) {
        cout << va_arg( args, int ) << endl;
        ++f;
    }
    va_end( args );
}
```

The previous example can be changed to the following code with corresponding changes to the contents of the virtual functions.

Example:

```
#include <iostream.h>
#include <stdarg.h>

struct B {
    void fun( char *f, ... )
    {
        va_list args;

        va_start( args, f );
        _fun( f, args );
        va_end( args );
    }
    virtual void _fun( char *, va_list );
};
~b
struct D : B {
    // this can be removed since using B::fun
    // will result in the same behaviour
    // since _fun is a virtual function
    void fun( char *f, ... )
    {
        va_list args;

        va_start( args, f );
        _fun( f, args );
        va_end( args );
    }
    virtual void _fun( char *, va_list );
};
~b
void B::_fun( char *f, va_list args )
{
    while( *f ) {
        cout << va_arg( args, char ) << endl;
        ++f;
    }
}
~b
void D::_fun( char *f, va_list args )
{
    while( *f ) {
        cout << va_arg( args, int ) << endl;
        ++f;
    }
}
```

```
~b
// no changes are required for users of the class
B x;
D y;

void dump( B *p )
{
    p->fun( "1234", 'a', 'b', 'c', 'd' );
    p->fun( "12", 'a', 'b' );
}

~b
void main()
{
    dump( &x );
    dump( &y );
}
```

506 *conversion of __based(void) pointer to virtual base class*

An __based(void) pointer to a class object cannot be converted to a pointer to virtual base class, since this conversion applies only to specific objects.

Example:

```
struct Base {};
struct Derived : virtual Base {};
Derived __based( void ) *p_derived;
Base __based( void ) *p_base = p_derived; // error
```

The conversion would be allowed if the base class were not virtual.

507 *class for target operand is not derived from class for source operand*

A member pointer conversion can only be performed safely when converting a base class member pointer to a derived class member pointer.

508 *conversion ambiguity: [pointer to class member] to [assignment object]*

The base class in the original member pointer is not a unique base class of the derived class.

509 *conversion of pointer to class member involves a private base class*

The member pointer conversion required access to a private base class. The access check did not succeed so the conversion is not allowed.

510 *conversion of pointer to class member involves a protected base class*

The member pointer conversion required access to a protected base class. The access check did not succeed so the conversion is not allowed.

511 *item is neither a non-static member function nor data member*

A member pointer can only be created for non-static member functions and non-static data members. Static members can have their address taken just like their file scope counterparts.

512 *function address cannot be converted to pointer to class member*

The indicated function address cannot be converted to pointer to class member.

513 *conversion ambiguity: [address of function] to [pointer to class member]*

The indicated conversion is ambiguous.

514 *addressed function is in a private base class*

The addressed function is in a private base class.

515 *addressed function is in a protected base class*

The addressed function is in a protected base class.

516 *class for object is not defined*

The left hand operand for the "." or ".*" operator must be of a class type that is completely defined.

Example:

```
class C;

int fun( C& x )
{
    return x.y;     // class C not defined
}
```

517 *left expression is not a class object*

The left hand operand for the ".*" operator must be of a class type since member pointers can only be used with classes.

518 *right expression is not a pointer to class member*

The right hand operand for the ".*" operator must be a member pointer type.

519 *cannot convert pointer to class of member pointer*

The class of the left hand operand cannot be converted to the class of the member pointer because it is not a derived class.

520 *conversion ambiguity: [pointer] to [class of pointer to class member]*

The class of the pointer to member is an ambiguous base class of the left hand operand.

521 *conversion of pointer to class of member pointer involves a private base class*

The class of the pointer to member is a private base class of the left hand operand.

522 *conversion of pointer to class of member pointer involves a protected base class*

The class of the pointer to member is a protected base class of the left hand operand.

523 *cannot convert object to class of member pointer*

The class of the left hand operand cannot be converted to the class of the member pointer because it is not a derived class.

524 *conversion ambiguity: [object] to [class object of pointer to class member]*

The class of the pointer to member is an ambiguous base class of the left hand operand.

525 *conversion of object to class of member pointer involves a private base class*

The class of the pointer to member is a private base class of the left hand operand.

526 *conversion of object to class of member pointer involves a protected base class*

The class of the pointer to member is a protected base class of the left hand operand.

527 *conversion of pointer to class member from a derived to a base class*

A member pointer can only be converted from a base class to a derived class. This is the opposite of the conversion rule for pointers.

528 *form is '#pragma inline_recursion en' where 'en' is 'on' or 'off'*

This ***pragma*** indicates whether inline expansion will occur for an inline function which is called (possibly indirectly) a subsequent time during an inline expansion. Either 'on' or 'off' must be specified.

529 *expression for number of array elements must be integral*

The expression for the number of elements in a ***new*** expression must be integral because it is used to calculate the size of the allocation (which is an integral quantity). The compiler will not automatically convert to an integer because of rounding and truncation issues with floating-point values.

530 *function accessed with '.*' or '->*' can only be called*

The result of the ".*" and "->*" operators can only be called because it is often specific to the instance used for the left hand operand.

531 *left operand must be a pointer, pointer to class member, or arithmetic*

The left operand must be a pointer, pointer to class member, or arithmetic.

532 *right operand must be a pointer, pointer to class member, or arithmetic*

The right operand must be a pointer, pointer to class member, or arithmetic.

533 *neither pointer to class member can be converted to the other*

The two member pointers being compared are from two unrelated classes. They cannot be compared since their members can never be related.

534 *left operand is not a valid pointer to class member*

The specified operator requires a pointer to member as the left operand.

Example:

```
struct S;
void fn( int S::* mp, int *p )
{
    if( p == mp )
        p[0] = 1;
}
```

535 *right operand is not a valid pointer to class member*

The specified operator requires a pointer to member as the right operand.

Example:

```
struct S;
void fn( int S::* mp, int *p )
{
    if( mp == p )
        p[0] = 1;
}
```

536 *cannot use '.*' nor '->*' with pointer to class member with zero value*

The compiler has detected a NULL pointer use with a member pointer dereference.

537 *operand is not a valid pointer to class member*

The operand cannot be converted to a valid pointer to class member.

Example:

```
struct S;
int S::* fn()
{
    int a;
    return a;
}
```

538 *destructor can be invoked only with '.' or '->'*

This is a restriction in the C++ language. An explicit invocation of a destructor is not recommended for objects that have their destructor called automatically.

539 *class of destructor must be class of object being destructed*

Destructors can only be called for the exact static type of the object being destroyed.

540 *destructor is not properly qualified*

An explicit destructor invocation can only be qualified with its own class.

541 *pointers to class members reference different object types*

Conversion of member pointers can only occur if the object types are identical. This is necessary to ensure type safety.

542 *operand must be pointer to class or struct*

The left hand operand of a '->*' operator must be a pointer to a class. This is a restriction in the C++ language.

543 *expression must have 'void' type*

If one operand of the ':' operator has ***void*** type, then the other operand must also have ***void*** type.

544 *expression types do not match for ':' operator*

The compiler could not bring both operands to a common type. This is necessary because the result of the conditional operator must be a unique type.

545 *cannot create an undefined type with 'operator new'*

A ***new*** expression cannot allocate an undefined type because it must know how large an allocation is required and it must also know whether there are any constructors to execute.

546 *delete of a pointer to an undefined type*

A ***delete*** expression cannot safely deallocate an undefined type because it must know whether there are any destructors to execute. In spite of this, the ISO/ANSI C++ Working Paper requires that an implementation support this usage.

Example:

```
struct U;

void foo( U *p, U *q ) {
    delete p;
    delete [] q;
}
```

547 *cannot access '%S' through a private base class*

The indicated symbol cannot be accessed because it requires access to a private base class.

548 *cannot access '%S' through a protected base class*

The indicated symbol cannot be accessed because it requires access to a protected base class.

549 *'sizeof' operand contains compiler generated information*

The type used in the 'sizeof' operand contains compiler generated information. Clearing a struct with a call to memset() would invalidate all of this information.

550 *cannot convert ':' operands to a common reference type*

The two reference types cannot be converted to a common reference type. This can happen when the types are not related through base class inheritance.

551 *conversion ambiguity: [reference to object] to [type of opposite ':' operand]*

One of the reference types is an ambiguous base class of the other. This prevents the compiler from converting the operand to a unique common type.

552 *conversion of reference to ':' object involves a private base class*

The conversion of the reference operands requires a conversion through a private base class.

553 *conversion of reference to ':' object involves a protected base class*

The conversion of the reference operands requires a conversion through a protected base class.

554 *expression must have type arithmetic, pointer, or pointer to class member*

This message means that the type cannot be converted to any of these types, also. All of the mentioned types can be compared against zero ('0') to produce a true or false value.

555 *expression for 'while' is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

556 *testing expression for 'for' is always false*

The compiler has detected that the expression will always be false. If this is not the expected behaviour, the code may contain a comparison of an unsigned value against zero (e.g., unsigned integers are always greater than or equal to zero). Comparisons against zero for addresses can also result in trivially false expressions.

557 *message number '%d' is invalid*

The message number used in the #pragma does not match the message number for any warning message. This message can also indicate that a number or '*' (meaning all warnings) was not found when it was expected.

558 *warning level must be an integer in range 0 to 9*

The new warning level that can be used for the warning can be in the range 0 to 9. The level 0 means that the warning will be treated as an error (compilation will not succeed). Levels 1 up to 9 are used to classify warnings. The -w option sets an upper limit on the level for warnings. By setting the level above the command line limit, you effectively ignore all cases where the warning shows up.

559 *function '%S' cannot be defined because it is generated by the compiler*

The indicated function cannot be defined because it is generated by the compiler. The compiler will automatically generate default constructors, copy constructors, assignment operators, and destructors according to the rules of the C++ language. This message indicates that you did not declare the function in the class definition.

560 *neither environment variable nor file found for '@' name*

The indirection operator for the command line will first check for an environment variable of the name and use the contents for the command line. If an environment variable is not found, a check for a file with the same name will occur.

561 *more than 5 indirections during command line processing*

The Watcom C++ compiler only allows a fixed number nested indirections using files or environment variables, to prevent runaway chains of indirections.

562 *cannot take address of non-static member function*

The only way to create a value that described the non-static member function is to use a member pointer.

563 *cannot generate default '%S' because class contains either a constant or a reference member*

An assignment operator cannot be generated because the class contains members that cannot be assigned into.

564 *cannot convert pointer to non-constant or volatile objects to 'const void*'*

A pointer to non-constant or volatile objects cannot be converted to 'const void*'.

565 *cannot convert pointer to non-constant or non-volatile objects to 'const volatile void*'*

A pointer to non-constant or non-volatile objects cannot be converted to 'const volatile void*'.

566 *cannot initialize pointer to non-volatile with a pointer to volatile*

A pointer to a non-volatile type cannot be initialized with a pointer to a volatile type because this would allow volatile data to be modified without volatile semantics via the non-volatile pointer to it.

567 *cannot pass a pointer or reference to a volatile object*

A pointer or reference to a volatile object cannot be passed in this context.

568 *cannot return a pointer or reference to a volatile object*

A pointer or reference to a volatile object cannot be returned.

569 *left expression is not a pointer to a volatile object*

One cannot assign a pointer to a volatile type to a pointer to a non-volatile type. This would allow a volatile object to be modified via the non-volatile pointer. Use a cast if this is absolutely necessary.

570 *virtual function override for '%S' is ambiguous*

This message indicates that there are at least two overrides for the function in the base class. The compiler cannot arbitrarily choose one so it is up to the programmer to make sure there is an unambiguous choice. Two of the overriding functions are listed as informational messages.

571 *initialization priority must be number 0-255, 'library', or 'program'*

An incorrect module initialization priority has been provided. Check the User's Guide for the correct format of the priority directive.

572 *previous 'case' label defined %L*

This informational message indicates where a preceding ***case*** label is defined.

573 *previous 'default' label defined %L*

This informational message indicates where a preceding ***default*** label is defined.

574 *label defined %L*

This informational message indicates where a label is defined.

575 *label referenced %L*

This informational message indicates where a label is referenced.

576 *object thrown has type: %T*

This informational message indicates the type of the object being thrown.

577 *object thrown has an ambiguous base class %T*

It is illegal to throw an object with a base class to which a conversion would be ambiguous.

Example:

```
struct ambiguous{ };
struct base1 : public ambiguous { };
struct base2 : public ambiguous { };
struct derived : public base1, public base2 { };

foo( derived &object )
{
    throw object;
}
```

The ***throw*** will cause an error to be displayed because an object of type "derived" cannot be converted to an object of type "ambiguous".

578 *form is '#pragma inline_depth level' where 'level' is 0 to 255*

This ***pragma*** sets the number of times inline expansion will occur for an inline function which contains calls to inline functions. The level must be a number from zero to 255. When the level is zero, no inline expansion occurs.

579 *pointer or reference truncated by cast*

The cast expression causes a conversion of a pointer value to another pointer value of smaller size. This can be caused by ***__near*** or ***__far*** qualifiers (i.e., casting a ***far*** pointer to a ***near*** pointer). Function pointers can also have a different size than data pointers in certain memory models. Because this message indicates that some information is being lost, check the code carefully.

580 *cannot find a constructor for given initializer argument list*

The initializer list provided for the ***new*** expression does not uniquely identify a single constructor.

581 *variable '%N' can only be based on a string in this context*

All of the based modifiers can only be applied to pointer types. The only based modifier that can be applied to non-pointer types is the '__based(__segname("WATCOM"))' style.

582 *memory model modifiers are not allowed for class members*

Class members describe the arrangement and interpretation of memory and, as such, assume the memory model of the address used to access the member.

583 *redefinition of the typedef name '%S' ignored*

The compiler has detected that a slightly different type has been assigned to a typedef name. The type is functionally equivalent but typedef redefinitions should be precisely identical.

584 *constructor for variable '%S' cannot be bypassed*

The variable may not be constructed when code is executing at the position the message indicated. The C++ language places these restrictions to prevent the use of unconstructed variables.

585 *syntax error; missing start of function body after constructor initializer*

Member initializers can only be used in a constructor's definition.

Example:

```
struct S {
    int a;
    S( int x = 1 ) : a(x)
    {
    }
};
```

586 *conversion ambiguity: [expression] to [type of default argument]*

A conversion to an ambiguous base class was detected in the default argument expression.

587 *conversion of expression for default argument is impossible*

A conversion to a unrelated class was detected in the default argument expression.

588 *syntax error before template name '%s'*

The identifier in the error message has been declared as a template name at this point in the code. This may be the cause of the syntax error.

589 *private base class accessed to convert default argument*

A conversion to a private base class was detected in the default argument expression.

590 *protected base class accessed to convert default argument*

A conversion to a protected base class was detected in the default argument expression.

591 *operand must be an 'lvalue' (cast produces 'rvalue')*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

592 *left operand must be an 'lvalue' (cast produces 'rvalue')*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

593 *right operand must be an 'lvalue' (cast produces 'rvalue')*

The compiler is expecting a value which can be assigned into. The result of a cast cannot be assigned into because a brand new value is always created. Assigning a new value to a temporary is a meaningless operation.

594 *construct resolved as a declaration/type*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been resolved in a certain direction. In this case, the construct has been determined to be part of a type. The final resolution varies between compilers so it is wise to change the source code so that the construct is not ambiguous. This is especially important in cases where the resolution is more than three tokens away from the start of the ambiguity.

595 *construct resolved as an expression*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been resolved in a certain direction. In this case, the construct has been determined to be part of an expression (a function-like cast). The final resolution varies between compilers so it is wise to change the source code so that the construct is not ambiguous. This is especially important in cases where the resolution is more than three tokens away from the start of the ambiguity.

596 *construct cannot be resolved*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct could not be resolved by the compiler. Please report this to Watcom so that the problem can be analysed.

597 *encountered another ambiguous construct during disambiguation*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that another ambiguous construct was found inside an ambiguous construct. The compiler will correctly disambiguate the construct. The programmer is advised to change code that exhibits this warning because this is definitely uncharted territory in the C++ language.

598 *ellipsis (...) argument contains compiler generated information*

A class with virtual functions or virtual bases is being passed to a function that will not know the type of the argument. Since this information can be encoded in a variety of ways, the code may not be portable to another environment.

Example:

```
struct S
{   virtual int foo();
};

static S sv;

extern int bar( S, ... );

static int test = bar( sv, 14, 64 );
```

The call to "bar" causes a warning, since the structure S contains information associated with the virtual function for that class.

599 *cannot convert argument for ellipsis (...) argument*

This argument cannot be used as an ellipsis (...) argument to a function.

600 *conversion ambiguity: [argument] to [ellipsis (...) argument]*

A conversion ambiguity was detected while converting an argument to an ellipsis (...) argument.

601 *converted function type has different #pragma from original function type*

Since a #pragma can affect calling conventions, one must be very careful performing casts involving different calling conventions.

602 *class value used as return value or argument in converted function type*

The compiler has detected a cast between "C" and "C++" linkage function types. The calling conventions are different because of the different language rules for copying structures.

603 *class value used as return value or argument in original function type*

The compiler has detected a cast between "C" and "C++" linkage function types. The calling conventions are different because of the different language rules for copying structures.

604 *must look ahead to determine whether construct is a declaration/type or an expression*

The C++ language contains language ambiguities that force compilers to rely on extra information in order to understand certain language constructs. The extra information required to disambiguate the language can be deduced by looking ahead in the source file. Once a single interpretation has been found, the compiler can continue analysing source code. See the ARM p.93 for more details. This warning is intended to inform the programmer that an ambiguous construct has been used. The final resolution varies between compilers so it is wise to change the source code so that the construct is not ambiguous.

605 *assembler: '%s'*

An error has been detected by the #pragma inline assembler.

606 *default argument expression cannot reference 'this'*

The order of evaluation for function arguments is unspecified in the C++ language document. Thus, a default argument must be able to be evaluated before the 'this' argument (or any other argument) is evaluated.

607 *#pragma aux must reference a "C" linkage function '%S'*

The method of assigning pragma information via the #pragma syntax is provided for compatibility with Watcom C. Because C only allows one function per name, this was adequate for the C language. Since C++ allows functions to be overloaded, a new method of referencing pragmas has been introduced.

Example:

```
#pragma aux this_in_SI parm caller [si] [ax];

struct S {
    void __pragma("this_in_SI") foo( int );
    void __pragma("this_in_SI") foo( char );
};
```

608 *assignment is ambiguous for operands used*

An ambiguity was detected while attempting to convert the right operand to the type of the left operand.

Example:

```
struct S1 {
    int a;
};

struct S2 : S1 {
    int b;
};

struct S3 : S2, S1 {
    int c;
};

S1* fn( S3 *p )
{
    return p;
}
```

In the example, ***class*** `S1` occurs ambiguously for an object or pointer to an object of type `S3.` A pointer to an `S3` object cannot be converted to a pointer to an `S1` object.

609 *pragma name '%s' is not defined*

Pragmas are defined with the #pragma aux syntax. See the User's Guide for the details of defining a pragma name. If the pragma has been defined then check the spelling between the definition and the reference of the pragma name.

610 *'%S' could not be generated by the compiler*

An error occurred while the compiler tried to generate the specified function. The error prevented the compiler from generating the function properly so the compilation cannot continue.

611 *'catch' does not immediately follow a 'try' or 'catch'*

The catch handler syntax must be used in conjunction with a try block.

Example:

```
void f()
{
    try {
        // code that may throw an exception
    } catch( int x ) {
        // handle 'int' exceptions
    } catch( ... ) {
        // handle all other exceptions
    }
}
```

612 *preceding 'catch' specified '...'*

Since an ellipsis "..." catch handler will handle any type of exception, no further catch handlers can exist afterwards because they will never execute. Reorder the catch handlers so that the "..." catch handler is the last handler.

613 *argument to extern "C" function contains compiler generated information*

A class with virtual functions or virtual bases is being passed to a function that will not know the type of the argument. Since this information can be encoded in a variety of ways, the code may not be portable to another environment.

Example:

```
struct S
{   virtual int foo();
};

static S sv;

extern "C" int bar( S );

static int test = bar( sv );
```

The call to "bar" causes a warning, since the structure S contains information associated with the virtual function for that class.

614 *previous try block defined %L*

This informational message indicates where a preceding ***try*** block is defined.

615 *previous catch block defined %L*

This informational message indicates where a preceding ***catch*** block is defined.

616 *'catch' handler can never be invoked*

Because the handlers for a ***try*** block are tried in order of appearance, the type specified in a preceding ***catch*** can ensure that the current handler will never be invoked. This occurs when a base class (or reference) precedes a derived class (or reference); when a pointer to a base class (or reference to the pointer) precedes a pointer to a derived class (or reference to the pointer); or, when "void*" or "void*&" precedes a pointer or a reference to the pointer.

Example:

```
struct BASE {};
struct DERIVED : public BASE {};

foo()
{
    try {
        // code for try
    } catch( BASE b ) {     // [1]
        // code
    } catch( DERIVED ) {    // warning: [1]
        // code
    } catch( BASE* pb ) {   // [2]
        // code
    } catch( DERIVED* pd ) {// warning: [2]
        // code
    } catch( void* pv ) {   // [3]
        // code
    } catch( int* pi ) {    // warning: [3]
        // code
    } catch( BASE& br ) {   // warning: [1]
        // code
    } catch( float*& pfr ) {// warning: [3]
        // code
    }
}
```

Each erroneous catch specification indicates the preceding catch block which caused the error.

617 *cannot overload extern "C" functions (the other function is '%S')*

The C++ language only allows you to overload functions that are strictly C++ functions. The compiler will automatically generate the correct code to distinguish each particular function based on its argument types. The extern "C" linkage mechanism only allows you to define one "C" function of a particular name because the C language does not support function overloading.

618 *function will be overload ambiguous with '%S' using default arguments*

The declaration declares a function that is indistinguishable from another function of the same name with default arguments.

Example:

```
void fn( int, int = 1 );
void fn( int );
```

Calling the function 'fn' with one argument is ambiguous because it could match either the first 'fn' with a default argument applied or the second 'fn' without any default arguments.

619 *linkage specification is different than previous declaration '%S'*

The linkage specification affects the binding of names throughout a program. It is important to maintain consistency because subtle problems could arise when the incorrect function is called. Usually this error prevents an unresolved symbol error during linking because the name of a declaration is affected by its linkage specification.

Example:

```
extern "C" void fn( void );
void fn( void )
{
}
```

620 *not enough segment registers available to generate '%s'*

Through a combination of options, the number of available segment registers is too small. This can occur when too many segment registers are pegged. This can be fixed by changing the command line options to only peg the segment registers that must absolutely be pegged.

621 *pure virtual destructors must have a definition*

This is an anomaly for pure virtual functions. A destructor is the only special function that is inherited and allowed to be virtual. A derived class must be able to call the base class destructor so a pure virtual destructor must be defined in a C++ program.

622 *jump into try block*

Jumps cannot enter ***try*** blocks.

Example:

```
foo( int a )
{
    if(a) goto tr_lab;

    try {
tr_lab:
        throw 1234;
    } catch( int ) {
        if(a) goto tr_lab;
    }

    if(a) goto tr_lab;
}
```

All the preceding goto's are illegal. The error is detected at the label for forward jumps and at the goto's for backward jumps.

623 *jump into catch handler*

Jumps cannot enter ***catch*** handlers.

Example:

```
foo( int a )
{
    if(a)goto ca_lab;

    try {
        if(a)goto ca_lab;
    } catch( int ) {
ca_lab:
    }

    if(a)goto ca_lab;
}
```

All the preceding goto's are illegal. The error is detected at the label for forward jumps and at the goto's for backward jumps.

624 *catch block does not immediately follow try block*

At least one ***catch*** handler must immediately follow the "}" of a ***try*** block.

Example:

```
extern void goop();
void foo()
{
    try {
        goop();
    }           // a catch block should follow!
}
```

In the example, there were no catch blocks after the ***try*** block.

625 *exceptions must be enabled to use feature (use 'xs' option)*

Exceptions are enabled by specifying the 'xs' option when the compiler is invoked. The error message indicates that a feature such as ***try, catch, throw,*** or function exception specification has been used without enabling exceptions.

626 *I/O error reading '%s': %s"*

When attempting to read data from a source or header file, the indicated system error occurred. Likely there is a hardware problem, or the file system has become corrupt.

627 *text following pre-processor directive*

A ***#else*** or ***#endif*** directive was found which had tokens following it rather than an end of line. Some UNIX style preprocessors allowed this, but it is not legal under standard C or C++. Make the tokens into a comment.

628 *expression is not meaningful*

This message indicates that the indicated expression is not meaningful. An expression is meaningful when a function is invoked, when an assignment or initialization is performed, or when the expression is casted to void.

Example:

```
void foo( int i, int j )
{
    i + j;  // not meaningful
}
```

629 *expression has no side effect*

The indicated expression does not cause a side effect. A side effect is caused by invoking a function, by an assignment or an initialization, or by reading a ***volatile*** variable.

Example:

```
int k;
void foo( int i, int j )
{
    i + j,  // no side effect (note comma)
    k = 3;
}
```

630 *source conversion type is "%T"*

This informational message indicates the type of the source operand, for the preceding conversion diagnostic.

631 *target conversion type is "%T"*

This informational message indicates the target type of the conversion, for the preceding conversion diagnostic.

632 *redeclaration of '%S' has different attributes*

A function cannot be made ***virtual*** or pure ***virtual*** in a subsequent declaration. All properties of a function should be described in the first declaration of a function. This is especially important for member functions because the properties of a class are affected by its member functions.

Example:

```
struct S {
    void fun();
};

virtual void S::fun()
{
}
```

633 *template class instantiation for '%T' was %L*

This informational message indicates that the error or warning was detected during the instantiation of a class template. The final type of the template class is shown as well as the location in the source where the instantiation was initiated.

634 *template function instantiation for '%S' was %L*

This informational message indicates that the error or warning was detected during the instantiation of a function template. The final type of the template function is shown as well as the location in the source where the instantiation was initiated.

635 *template class member instantiation was %L*

This informational message indicates that the error or warning was detected during the instantiation of a member of a class template. The location in the source where the instantiation was initiated is shown.

636 *function template binding for '%S' was %L*

This informational message indicates that the error or warning was detected during the binding process of a function template. The binding process occurs at the point where arguments are analysed in order to infer what types should be used in a function template instantiation. The function template in question is shown along with the location in the source code that initiated the binding process.

637 *function template binding of '%S' was %L*

This informational message indicates that the error or warning was detected during the binding process of a function template. The binding process occurs at the point where a function prototype is analysed in order to see if the prototype matches any function template of the same name. The function template in question is shown along with the location in the source code that initiated the binding process.

638 *'%s' defined %L*

This informational message indicates where the class in question was defined. The message is displayed following an error or warning diagnostic for the class in question.

Example:

```
class S;
int foo( S*p )
{
    return p->x;
}
```

The variable `p` is a pointer to an undefined class and so will cause an error to be generated. Following the error, the informational message indicates the line at which the class S was declared.

639 *form is '#pragma template_depth level' where 'level' is a non-zero number*

This ***pragma*** sets the number of times templates will be instantiated for nested instantiations. The depth check prevents infinite compile times for incorrect programs.

640 *possible non-terminating template instantiation (use "#pragma template_depth %d" to increase depth)*

This message indicates that a large number of expansions were required to complete a template class or template function instantiation. This may indicate that there is an erroneous use of a template. If the program will complete given more depth, try using the suggested #pragma in the error message to increase the depth. The number provided is double the previous value.

641 *cannot inherit a partially defined base class '%T'*

This message indicates that the base class was in the midst of being defined when it was inherited. The storage requirements for a ***class*** type must be known when inheritance is involved because the layout of the final class depends on knowing the complete contents of all base classes.

Example:

```
struct Partial {
    struct Nested : Partial {
        int n;
    };
};
```

642 *ambiguous function: %F defined %L*

This informational message shows the functions that were detected to be ambiguous.

Example:

```
int amb( char );            // will be ambiguous
int amb( unsigned char );   // will be ambiguous
int amb( char, char );
int k = amb( 14 );
```

The constant value 14 has an ***int*** type and so the attempt to invoke the function `amb` is ambiguous. The first two functions are ambiguous (and will be displayed); the third is not considered (nor displayed) since it is declared to have a different number of arguments.

643 *cannot convert argument %d defined %L*

This informational message indicates the first argument which could not be converted to the corresponding type for the declared function. It is displayed when there is exactly one function declared with the indicated name.

644 *'this' cannot be converted*

This informational message indicates the ***this*** pointer for the function which could not be converted to the type of the ***this*** pointer for the declared function. It is displayed when there is exactly one function declared with the indicated name.

645 *rejected function: %F defined %L*

This informational message shows the overloaded functions which were rejected from consideration during function-overload resolution. These functions are displayed when there is more than one function with the indicated name.

646 *'%T' operator can be used*

Following a diagnosis of operator ambiguity, this information message indicates that the operator can be applied with operands of the type indicated in the message.

Example:

```
struct S {
    S( int );
    operator int();
    S operator+( int );
};
S s(15);
int k = s + 123;    // "+" is ambiguous
```

In the example, the "+" operation is ambiguous because it can implemented as by the addition of two integers (with `S::operator int` applied to the second operand) or by a call to `S::operator+`. This informational message indicates that the first is possible.

647 *cannot #undef '%s'*

The predefined macros `__cplusplus`, `__DATE__`, `__FILE__`, `__LINE__`, `__STDC__`, and `__TIME__` cannot be undefined using the ***#undef*** directive.

Example:

```
#undef __cplusplus
#undef __DATE__
#undef __FILE__
#undef __LINE__
#undef __STDC__
#undef __TIME__
```

All of the preceding directives are not permitted.

648 *cannot #define '%s'*

The predefined macros `__cplusplus`, `__DATE__`, `__FILE__`, `__LINE__`, `__STDC__`, and `__TIME__` cannot be defined using the ***#define*** directive.

Example:

```
#define __cplusplus 1
#define __DATE__    2
#define __FILE__    3
#define __LINE__    4
#define __STDC__    5
#define __TIME__    6
```

All of the preceding directives are not permitted.

649 *template function '%F' defined %L*

This informational message indicates where the function template in question was defined. The message is displayed following an error or warning diagnostic for the function template in question.

Example:

```
template <class T>
    void foo( T, T * )
    {
    }

void bar()
{
    foo(1);         // could not instantiate
}
```

The function template for `foo` cannot be instantiated for a single argument causing an error to be generated. Following the error, the informational message indicates the line at which `foo` was declared.

650 *ambiguous function template: %F defined %L*

This informational message shows the function templates that were detected to be ambiguous for the arguments at the call point.

651 *cannot instantiate %S*

This message indicates that the function template could not be instantiated for the arguments supplied. It is displayed when there is exactly one function template declared with the indicated name.

652 *rejected function template: %F defined %L*

This informational message shows the overloaded function template which was rejected from consideration during function-overload resolution. These functions are displayed when there is more than one function or function template with the indicated name.

653 *operand cannot be a function*

The indicated operation cannot be applied to a function.

Example:

```
int Fun();
int j = ++Fun;  // illegal
```

In the example, the attempt to increment a function is illegal.

654 *left operand cannot be a function*

The indicated operation cannot be applied to the left operand which is a function.

Example:

```
extern int Fun();
void foo()
{
    Fun = 0;    // illegal
}
```

In the example, the attempt to assign zero to a function is illegal.

655 *right operand cannot be a function*

The indicated operation cannot be applied to the right operand which is a function.

Example:

```
extern int Fun();
void foo()
{
    void* p = 3[Fun];   // illegal
}
```

In the example, the attempt to subscript a function is illegal.

656 *define this function inside its class definition (may improve code quality)*

The Watcom C++ compiler has found a constructor or destructor with an empty function body. An empty function body can usually provide optimization opportunities so the compiler is indicating that by defining the function inside its class definition, the compiler may be able to perform some important optimizations.

Example:

```
struct S {
    ~S();
};

S::~S() {
}
```

657 *define this function inside its class definition (could have improved code quality)*

The Watcom C++ compiler has found a constructor or destructor with an empty function body. An empty function body can usually provide optimization opportunities so the compiler is indicating that by defining the function inside its class definition, the compiler may be able to perform some important optimizations. This particular warning indicates that the compiler has already found an opportunity in previous code but it found out too late that the constructor or destructor had an empty function body.

Example:

```
struct S {
    ~S();
};
struct T : S {
    ~T() {}
};

S::~S() {
}
```

658 *cannot convert address of overloaded function '%S'*

This information message indicates that an address of an overloaded function cannot be converted to the indicated type.

Example:

```
int ovload( char );
int ovload( float );
int routine( int (*)( int );
int k = routine( ovload );
```

The first argument for the function `routine` cannot be converted, resulting in the informational message.

659 *expression cannot have 'void' type*

The indicated expression cannot have a ***void*** type.

Example:

```
main( int argc, char* argv )
{
    if( (void)argc ) {
        return 5;
    } else {
        return 9;
    }
}
```

Conditional expressions, such as the one illustrated in the ***if*** statement cannot have a ***void*** type.

660 *cannot reference a bit field*

The smallest addressable unit is a byte. You cannot reference a bit field.

Example:

```
struct S
{   int bits :6;
    int bitfield :10;
};
S var;
int& ref = var.bitfield;    // illegal
```

661 *cannot assign to object having an undefined class*

An assignment cannot be be made to an object whose class has not been defined.

Example:

```
class X;            // declared, but not defined
extern X& foo();    // returns reference (ok)
extern X obj;
void goop()
{
    obj = foo();    // error
}
```

662 *cannot create member pointer to constructor*

A member pointer value cannot reference a constructor.

Example:

```
class C {
    C();
};
int foo()
{
    return 0 == &C::C;
}
```

663 *cannot create member pointer to destructor*

A member pointer value cannot reference a destructor.

Example:

```
class C {
    ~C();
};
int foo()
{
    return 0 == &C::~C;
}
```

664 *attempt to initialize a non-constant reference with a temporary object*

A temporary value cannot be converted to a non-constant reference type.

Example:

```
struct C {
    C( C& );
    C( int );
};

C & c1 = 1;
C c2 = 2;
```

The initializations of `c1` and `c2` are erroneous, since temporaries are being bound to non-const references. In the case of `c1,` an implicit constructor call is required to convert the integer to the correct object type. This results in a temporary object being created to initialize the reference. Subsequent code can modify this temporary's state. The initialization of `c2,` is erroneous for a similar reason. In this case, the temporary is being bound to the non-const reference argument of the copy constructor.

665 *temporary object used to initialize a non-constant reference*

Ordinarily, a temporary value cannot be bound to a non-constant reference. There is enough legacy code present that the Watcom C++ compiler issues a warning in cases that should be errors. This may change in the future so it is advisable to correct the code as soon as possible.

666 *assuming unary 'operator &' not overloaded for type '%T'*

An explicit address operator can be applied to a reference to an undefined class. The Watcom C++ compiler will assume that the address is required but it does not know whether this was the programmer's intention because the class definition has not been seen.

Example:

```
struct S;

S * fn( S &y ) {
    // assuming no operator '&' defined
    return &y;
}
```

667 *'va_start' macro will not work without an argument before "..."*

The warning indicates that it is impossible to access the arguments passed to the function without declaring an argument before the "..." argument. The "..." style of argument list (without any other arguments) is only useful as a prototype or if the function is designed to ignore all of its arguments.

Example:

```
void fn( ... )
{
}
```

668 *'va_start' macro will not work with a reference argument before "..."*

The warning indicates that taking the address of the argument before the "..." argument, which 'va_start' does in order to access the variable list of arguments, will not give the expected result. The arguments will have to be rearranged so that an acceptable argument is declared before the "..." argument or a dummy ***int*** argument can be inserted after the reference argument with the corresponding adjustments made to the callers of the function.

Example:

```
#include <stdarg.h>

void fn( int &r, ... )
{
    va_list args;

    // address of 'r' is address of
    // object 'r' references so
    // 'va_start' will not work properly
    va_start( args, r );
    va_end( args );
}
```

669 *'va_start' macro will not work with a class argument before "..."*

This warning is specific to C++ compilers that quietly convert class arguments to class reference arguments. The warning indicates that taking the address of the argument before the "..." argument, which 'va_start' does in order to access the variable list of arguments, will not give the expected result. The arguments will have to be rearranged so that an acceptable argument is declared before the "..." argument or a dummy ***int*** argument can be inserted after the class argument with the corresponding adjustments made to the callers of the function.

Example:

```
#include <stdarg.h>

struct S {
    S();
};

void fn( S c, ... )
{
    va_list args;

    // Watcom C++ passes a pointer to
    // the temporary created for passing
    // 'c' rather than pushing 'c' on the
    // stack so 'va_start' will not work
    // properly
    va_start( args, c );
    va_end( args );
}
```

670 *function modifier conflicts with previous declaration '%S'*

The symbol declaration conflicts with a previous declaration with regard to function modifiers. Either the previous declaration did not have a function modifier or it had a different one.

Example:

```
#pragma aux never_returns aborts;

void fn( int, int );
void __pragma("never_returns") fn( int, int );
```

671 *function modifier cannot be used on a variable*

The symbol declaration has a function modifier being applied to a variable or non-function. The cause of this may be a declaration with a missing function argument list.

Example:

```
int (* __pascal ok)();
int (* __pascal not_ok);
```

672 *'%T' contains the following pure virtual functions*

This informational message indicates that the class contains pure virtual function declarations. The class is definitely abstract as a result and cannot be used to declare variables. The pure virtual functions declared in the class are displayed immediately following this message.

Example:

```
struct A {
    void virtual fn( int ) = 0;
};

A x;
```

673 *'%T' has no implementation for the following pure virtual functions*

This informational message indicates that the class is derived from an abstract class but the class did not override enough virtual function declarations. The pure virtual functions declared in the class are displayed immediately following this message.

Example:

```
struct A {
    void virtual fn( int ) = 0;
};
struct D : A {
};

D x;
```

674 *pure virtual function '%F' defined %L*

This informational message indicates that the pure virtual function has not been overridden. This means that the class is abstract.

Example:

```
struct A {
    void virtual fn( int ) = 0;
};
struct D : A {
};

D x;
```

675 *restriction: standard calling convention required for '%S'*

The indicated function may be called by the C++ run-time system using the standard calling convention. The calling convention specified for the function is incompatible with the standard convention. This message may result when `__pascal` is specified for a default constructor, a copy constructor, or a destructor. It may also result when `parm reverse` is specified in a ***#pragma*** for the function.

676 *number of arguments in function call is incorrect*

The number of arguments in the function call does not match the number declared for the function type.

Example:

```
extern int (*pfn)( int, int );
int k = pfn( 1, 2, 3 );
```

In the example, the function pointer was declared to have two arguments. Three arguments were used in the call.

677 *function has type '%T'*

This informational message indicates the type of the function being called.

678 *invalid octal constant*

The constant started with a '0' digit which makes it look like an octal constant but the constant contained the digits '8' and '9'. The problem could be an incorrect octal constant or a missing '.' for a floating constant.

Example:

```
int i = 0123456789;     // invalid octal constant
double d = 0123456789;  // missing '.'?
```

679 *class template definition started %L*

This informational message indicates where the class template definition started so that any problems with missing braces can be fixed quickly and easily.

Example:

```
template <class T>
    struct S {
        void f1() {
        // error missing '}'
    };

template <class T>
    struct X {
        void f2() {
        }
    };
```

680 *constructor initializer started %L*

This informational message indicates where the constructor initializer started so that any problems with missing parenthesis can be fixed quickly and easily.

Example:

```
struct S {
    S( int x ) : a(x), b(x // missing parenthesis
    {
    }
};
```

681 *zero size array must be the last data member*

The language extension that allows a zero size array to be declared in a class definition requires that the array be the last data member in the class.

Example:

```
struct S {
    char a[];
    int b;
};
```

682 *cannot inherit a class that contains a zero size array*

The language extension that allows a zero size array to be declared in a class definition disallows the use of the class as a base class. This prevents the programmer from corrupting storage in derived classes through the use of the zero size array.

Example:

```
struct B {
    int b;
    char a[];
};
struct D : B {
    int d;
};
```

683 *zero size array '%S' cannot be used in a class with base classes*

The language extension that allows a zero size array to be declared in a class definition requires that the class not have any base classes. This is required because the C++ compiler must be free to organize base classes in any manner for optimization purposes.

Example:

```
struct B {
    int b;
};
struct D : B {
    int d;
    char a[];
};
```

684 *cannot catch abstract class object*

C++ does not allow abstract classes to be instantiated and so an abstract class object cannot be specified in a ***catch*** clause. It is permissible to catch a reference to an abstract class.

Example:

```
class Abstract {
public:
    virtual int foo() = 0;
};

class Derived : Abstract {
public:
    int foo();
};

int xyz;

void func( void ) {
    try {
        throw Derived();
    } catch( Abstract abstract ) {   // object
        xyz = 1;
    }
}
```

The catch clause in the preceding example would be diagnosed as improper, since an abstract class is specified. The example could be coded as follows.

Example:

```
class Abstract {
public:
    virtual int foo() = 0;
};

class Derived : Abstract {
public:
    int foo();
};

int xyz;

void func( void ) {
    try {
        throw Derived();
    } catch( Abstract & abstract ) {  // reference
        xyz = 1;
    }
}
```

685 *non-static member function '%S' cannot be specified*

The indicated non-static member function cannot be used in this context. For example, such a function cannot be used as the second or third operand of the conditional operator.

Example:

```
struct S {
    int foo();
    int bar();
    int fun();
};

int S::fun( int i ) {
    return (i ? foo : bar)();
}
```

Neither `foo` nor `bar` can be specified as shown in the example. The example can be properly coded as follows:

Example:

```
struct S {
    int foo();
    int bar();
    int fun();
};

int S::fun( int i ) {
    return i ? foo() : bar();
}
```

686 *attempt to convert pointer or reference from a base to a derived class*

A pointer or reference to a base class cannot be converted to a pointer or reference, respectively, of a derived class, unless there is an explicit cast. The `return` statements in the following example will be diagnosed.

Example:

```
struct Base {};
struct Derived : Base {};

Base b;

Derived* ReturnPtr() { return &b; }
Derived& ReturnRef() { return b; }
```

The following program would be acceptable:

Example:

```
struct Base {};
struct Derived : Base {};

Base b;

Derived* ReturnPtr() { return (Derived*)&b; }
Derived& ReturnRef() { return (Derived&)b; }
```

687 *expression for 'while' is always true*

The compiler has detected that the expression will always be true. Consequently, the loop will execute infinitely unless there is a ***break*** statement within the loop or a ***throw*** statement is executed while executing within the loop. If such an infinite loop is required, it can be coded as `for( ;)` without causing warnings.

688 *testing expression for 'for' is always true*

The compiler has detected that the expression will always be true. Consequently, the loop will execute infinitely unless there is a ***break*** statement within the loop or a ***throw*** statement is executed while executing within the loop. If such an infinite loop is required, it can be coded as `for( ;)` without causing warnings.

689 *conditional expression is always true (non-zero)*

The indicated expression is a non-zero constant and so will always be true.

690 *conditional expression is always false (zero)*

The indicated expression is a zero constant and so will always be false.

691 *expecting a member of '%T' to be defined in this context*

A class template member definition must define a member of the associated class template. The complexity of the C++ declaration syntax can make this error hard to identify visually.

Example:

```
template <class T>
    struct S {
        typedef int X;
        static X fn( int );
        static X qq;
    };

template <class T>
    S<T>::X fn( int ) {// should be 'S<T>::fn'

        return fn( 2 );
    }

template <class T>
    S<T>::X qq = 1;     // should be 'S<T>::q'

S<int> x;
```

692 *cannot throw an abstract class*

An abstract class cannot be thrown since copies of that object may have to be made (which is impossible);

Example:

```
struct abstract_class {
    abstract_class( int );
    virtual int foo() = 0;
};

void goop()
{
    throw abstract_class( 17 );
}
```

The ***throw*** expression is illegal since it specifies an abstract class.

693 *cannot create pre-compiled header file '%s'*

The compiler has detected a problem while trying to open the pre-compiled header file for write access.

694 *error occurred while writing pre-compiled header file*

The compiler has detected a problem while trying to write some data to the pre-compiled header file.

695 *error occurred while reading pre-compiled header file*

The compiler has detected a problem while trying to read some data from the pre-compiled header file.

696 *pre-compiled header file being recreated*

The existing pre-compiled header file may either be corrupted or is a version that the compiler cannot use due to updates to the compiler. A new version of the pre-compiled header file will be created.

697 *pre-compiled header file being recreated (different compile options)*

The compiler has detected that the command line options have changed enough so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

698 *pre-compiled header file being recreated (different #include file)*

The compiler has detected that the first ***#include*** file name is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

699 *pre-compiled header file being recreated (different current directory)*

The compiler has detected that the working directory is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

700 *pre-compiled header file being recreated (different INCLUDE path)*

The compiler has detected that the INCLUDE path is different so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

701 *pre-compiled header file being recreated ('%s' has been modified)*

The compiler has detected that an include file has changed so the contents of the pre-compiled header file cannot be used. A new version of the pre-compiled header file will be created.

702 *pre-compiled header file being recreated (macro '%s' is different)*

The compiler has detected that a macro definition is different so the contents of the pre-compiled header file cannot be used. The macro was referenced during processing of the header file that created the pre-compiled header file so the contents of the pre-compiled header may be affected. A new version of the pre-compiled header file will be created.

703 *pre-compiled header file being recreated (macro '%s' is not defined)*

The compiler has detected that a macro has not been defined so the contents of the pre-compiled header file cannot be used. The macro was referenced during processing of the header file that created the pre-compiled header file so the contents of the pre-compiled header may be affected. A new version of the pre-compiled header file will be created.

704 *command line specifies smart windows callbacks and DS not equal to SS*

An illegal combination of switches has been detected. The windows smart callbacks option cannot be combined with either of the build DLL or DS not equal to SS options.

705 *class '%N' cannot be used with #pragma dump_object_model*

The indicated name has not yet been declared or has been declared but not yet been defined as a class. Consequently, the object model cannot be dumped.

706 *repeated modifier is '%s'*

This informational message indicates what modifier was repeated in the declaration.

Example:

```
typedef int __far FARINT;
FARINT __far *p;        // repeated __far modifier
```

707 *semicolon (';') may be missing after class/enum definition*

This informational message indicates that a missing semicolon (';') may be the cause of the error.

Example:

```
struct S {
    int x,y;
    S( int, int );
} // missing semicolon ';'

S::S( int x, int y ) : x(x), y(y) {
}
```

708 *cannot return a type of unknown size*

A value of an unknown type cannot be returned.

Example:

```
class S;
S foo();

int goo()
{
    foo();
}
```

In the example, foo cannot be invoked because the class which it returns has not been defined.

709 *cannot initialize array member '%S'*

An array class member cannot be specified as a constructor initializer.

Example:

```
class S {
public:
    int arr[3];
    S();
};
S::S() : arr( 1, 2, 3 ) {}
```

In the example, `arr` cannot be specified as a constructor initializer. Instead, the array may be initialized within the body of the constructor.

Example:

```
class S {
public:
    int arr[3];
    S();
};
S::S()
{
    arr[0] = 1;
    arr[1] = 2;
    arr[2] = 3;
}
```

710 *file '%s' will #include itself forever*

The compiler has detected that the file in the message has been ***#include*** from within itself without protecting against infinite inclusion. This can happen if ***#ifndef*** and ***#define*** header file protection has not been used properly.

Example:

```
#include __FILE__
```

711 *'mutable' may only be used for non-static class members*

A declaration in file scope or block scope cannot have a storage class of ***mutable.***

Example:

```
mutable int a;
```

712 *'mutable' member cannot also be 'const'*

A ***mutable*** member can be modified even if its class object is ***const.*** Due to the semantics of ***mutable,*** the programmer must decide whether a member will be ***const*** or ***mutable*** because it cannot be both at the same time.

Example:

```
struct S {
    mutable const int * p;      // OK
    mutable int * const q;      // error
};
```

713 *left operand cannot be of type 'bool'*

The left hand side of an assignment operator cannot be of type ***bool*** except for simple assignment. This is a restriction required in the C++ language.

Example:

```
bool q;

void fn()
{
    q += 1;
}
```

714 *operand cannot be of type 'bool'*

The operand of both postfix and prefix "--" operators cannot be of type ***bool.*** This is a restriction required in the C++ language.

Example:

```
bool q;

void fn()
{
    --q;   // error
    q--;   // error
}
```

715 *member '%N' has not been declared in '%T'*

The compiler has found a member which has not been previously declared. The symbol may be spelled differently than the declaration, or the declaration may simply not be present.

Example:

```
struct X { int m; };

void fn( X *p )
{
    p->x = 1;
}
```

716 *integral value may be truncated*

This message indicates that the compiler knows that all values will not be preserved after the assignment or initialization. If this is acceptable, cast the value to the appropriate type in the assignment or initialization.

Example:

```
char inc( char c )
{
    return c + 1;
}
```

717 *left operand type is '%T'*

This informational message indicates the type of the left hand side of the expression.

718 *right operand type is '%T'*

This informational message indicates the type of the right hand side of the expression.

719 *operand type is '%T'*

This informational message indicates the type of the operand.

720 *expression type is '%T'*

This informational message indicates the type of the expression.

721 *virtual function '%S' cannot have its return type changed*

This restriction is due to the relatively new feature in the C++ language that allows return values to be changed when a virtual function has been overridden. It is not possible to support both features because in order to support changing the return value of a function, the compiler must construct a "wrapper" function that will call the virtual function first and then change the return value and return. It is not possible to do this with "..." style functions because the number of parameters is not known.

Example:

```
struct B {
};
struct D : virtual B {
};

struct X {
    virtual B *fn( int, ... );
};
struct Y : X {
    virtual D *fn( int, ... );
};
```

722 *__declspec('%N') is not supported*

The identifier used in the ***__declspec*** declaration modifier is not supported by Watcom C++.

723 *attempt to construct a far object when data model is near*

Constructors cannot be applied to objects which are stored in far memory when the default memory model for data is near.

Example:

```
struct Obj
{   char *p;
    Obj();
};

Obj far obj;
```

The last line causes this error to be displayed when the memory model is small (switch -ms), since the memory model for data is near.

724 *-zo is an obsolete switch (has no effect)*

The ***-zo*** option was required in an earlier version of the compiler but is no longer used.

725 *"%s"*

This is a user message generated with the ***#pragma message*** preprocessing directive.

Example:

```
#pragma message( "my very own warning" );
```

726 *no reference to formal parameter '%S'*

There are no references to the declared formal parameter. The simplest way to remove this warning in C++ is to remove the name from the argument declaration.

Example:

```
int fn1( int a, int b, int c )
{
    // 'b' not referenced
    return a + c;
}
int fn2( int a, int /* b */, int c )
{
    return a + c;
}
```

727 *cannot dereference a pointer to 'void'*

A pointer to ***void*** is used as a generic pointer but it cannot be dereferenced.

Example:

```
void fn( void *p )
{
    return *p;
}
```

728 *class modifiers for '%T' conflict with class modifiers for '%T'*

A conflict between class modifiers for classes related through inheritance has been detected. A conflict will occur if two base classes have class modifiers that are different. The conflict can be resolved by ensuring that all classes related through inheritance have the same class modifiers. The default resolution is to have no class modifier for the derived base.

Example:

```
struct __cdecl B1 {
    void fn( int );
};
struct __stdcall B2 {
    void fn( int );
};
struct D : B1, B2 {
};
```

729 *invalid hexadecimal constant*

The constant started with a ’0x’ prefix which makes it look like a hexadecimal constant but the constant was not followed by any hexadecimal digits.

Example:

```
unsigned i = 0x;     // invalid hex constant
```

730 *return type of ’operator ->’ will not allow ’->’ to be applied*

This restriction is a result of the transformation that the compiler performs when the ***operator ->*** is overloaded. The transformation involves transforming the expression to invoke the operator with "->" applied to the result of ***operator ->.*** This warning indicates that the ***operator ->*** can never be used as an overloaded operator. The only way the operator can be used is to explicitly call it by name.

Example:

```
struct S {
    int a;
    void *operator ->();
};

void *fn( S &q )
{
    return q.operator ->();
}
```

731 *class should have a name since it needs a constructor or a destructor*

The class definition does not have a class name but it includes members that have constructors or destructors. Since the class has C++ semantics, it should be have a name in case the constructor or destructor needs to be referenced.

Example:

```
struct P {
    int x,y;
    P();
};

typedef struct {
    P c;
    int v;
} T;
```

732 *class should have a name since it inherits a class*

The class definition does not have a class name but it inherits a class. Since the class has C++ semantics, it should be have a name in case the constructor or destructor needs to be referenced.

Example:

```
struct P {
    int x,y;
    P();
};

typedef struct : P {
    int v;
} T;
```

733 *cannot open pre-compiled header file '%s'*

The compiler has detected a problem while trying to open the pre-compiled header file for read/write access.

734 *invalid second argument to 'va_start'*

The second argument to the va_start macro should be the name of the argument just before the "..." in the argument list.

735 *'//' style comment continues on next line*

The compiler has detected a line continuation during the processing of a C++ style comment ("//"). The warning can be removed by switching to a C style comment ("/**/"). If you require the comment to be terminated at the end of the line, make sure that the backslash character is not the last character in the line.

Example:

```
#define XX 23 // comment start \
comment \
end

int x = XX; // comment start ...\
comment end
```

736 *cannot open file '%s' for write access*

The compiler has detected a problem while trying to open the indicated file for write access.

737 *implicit conversion of pointers to integral types of same size*

The compiler allows, when extensions are enabled, implicit conversions between pointers to integral types when the size of the integral types are the same. Thus, conversions from ***unsigned char*** to either ***char*** or ***signed char*** would be allowed. This is an extension as the ISO/ANSI Draft Working Paper permits implicit conversions only when the types pointed at are identical.

According to the ISO/ANSI Draft Working Paper, a string literal is an array of ***char.*** Consequently, it is illegal to initialize or assign the pointer resulting from that literal to a pointer of either ***unsigned char*** or ***signed char,*** since these pointers point at objects of a different type. When extensions are enabled, this condition is diagnosed as a warning; otherwise, it is an error.

738 *option requires a number*

The specified option is not recognized by the compiler since there was no number after it (i.e., "-w=1"). Numbers must be non-negative decimal numbers.

739 *option -fc specified more than once*

The -fc option can be specified at most once on a command line.

740 *option -fc specified in batch file of commands*

The -fc option cannot be specified on a line in the batch file of command lines specified by the -fc option on the command line used to invoke the compiler.

741 *file specified by -fc is empty or cannot be read*

The file specified using the -fc option is either empty or an input/output error was diagnosed for the file.

742 *cannot open file specified by -fc option*

The compiler was unable to open the indicated file. Most likely, the file does not exist. An input/output error is also possible.

743 *input/output error reading the file specified by -fc option*

The compiler was unable to open the indicated file. Most likely, the file does not exist. An input/output error is also possible.

744 *'%N' does not have a return type specified ('int' assumed)*

In C++, operator functions should have an explicit return type specified. In future revisions of the ISO/ANSI C++ standard, the use of default int type specifiers may be prohibited so removing any dependencies on default int early will prevent problems in the future.

Example:

```
struct S {
    operator = ( S const & );
    operator += ( S const & );
};
```

745 *cannot initialize reference to non-constant with a constant object*

A reference to a non-constant object cannot be initialized with a reference to a constant type because this would allow constant data to be modified via the non-constant pointer to it.

Example:

```
extern const int *pic;
extern int & ref = pic;
```

746 *processing %s*

This informational message indicates where an error or warning was detected while processing the switches specified on the command line, in environment variables, in command files (using the '@' notation), or in the batch command file (specified using the -fc option).

747 *'class %T' has not been defined*

This informational message indicates a class which was not defined. This is noted following an error or warning message because it often helps to a user to determine the cause of that diagnostic.

748 *cannot catch undefined class object*

C++ does not allow abstract classes to be copied and so an undefined class object cannot be specified in a ***catch*** clause. It is permissible to catch a reference to an undefined class.

749 *class '%T' cannot be used since its definition has errors*

The analysis of the expression could not continue due to previous errors diagnosed in the class definition.

750 *function prototype in block scope missing 'extern'*

This warning can be triggered when the intent is to define a variable with a constructor. Due to the complexities of parsing C++, statements that appear to be variable definitions may actually parse as a function prototype. A work-around for this problem is contained in the example. If a prototype is desired, add the ***extern*** storage class to remove this warning.

Example:

```
struct C {
};
struct S {
    S( C );
};
void foo()
{
    S a( C() ); // function prototype!
    S b( (C()) );// variable definition

    int bar( int );// warning
    extern int sam( int ); // no warning
}
```

751 *function prototype is '%T'*

This informational message indicates what the type of the function prototype is for the message in question.

752 *class '%T' contains a zero size array*

This warning is triggered when a class with a zero sized array is used in an array or as a class member. This is a questionable practice since a zero sized array at the end of a class often indicates a class that is dynamically sized when it is constructed.

Example:

```
struct C {
    C *next;
    char name[];
};

struct X {
    C q;
};

C a[10];
```

753 *invalid 'new' modifier*

The Watcom C++ compiler does not support new expression modifiers but allows them to match the ambient memory model for compatibility. Invalid memory model modifiers are also rejected by the compiler.

Example:

```
int *fn( unsigned x )
{
    return new __interrupt int[x];
}
```

754 *'__declspec(thread)' data '%S' must be link-time initialized*

This error message indicates that the data item in question either requires a constructor, destructor, or run-time initialization. This cannot be supported for thread-specific data at this time.

Example:

```
#include <stdlib.h>

struct C {
    C();
};
struct D {
    ~D();
};

C __declspec(thread) c;
D __declspec(thread) d;
int __declspec(thread) e = rand();
```

755 *code may not work properly if this module is split across a code segment*

The "zm" option allows the compiler to generate functions into separate segments that have different names so that more than 64k of code can be generated in one object file. Unfortunately, if an explicit near function is coded in a large code model, the possibility exists that the linker can place the near function in a separate code segment than a function that calls it. This would cause a linker error followed by an execution error if the executable is executed. The "zmf" option can be used if you require explicit near functions in your code.

Example:

```
// These functions may not end up in the
// same code segment if the -zm option
// is used. If this is the case, the near
// call will not work since near functions
// must be in the same code segment to
// execute properly.
static int near near_fn( int x )
{
    return x + 1;
}

int far_fn( int y )
{
    return near_fn( y * 2 );
}
```

756 *#pragma extref: symbol '%N' not declared*

This error message indicates that the symbol referenced by ***#pragma extref*** has not been declared in the context where the pragma was encountered.

757 *#pragma extref: overloaded function '%S' cannot be used*

An external reference can be emitted only for external functions which are not overloaded.

758 *#pragma extref: '%N' is not a function or data*

This error message indicates that the symbol referenced by ***#pragma extref*** cannot have an external reference emitted for it because the referenced symbol is neither a function nor a data item. An external reference can be emitted only for external functions which are not overloaded and for external data items.

759 *#pragma extref: '%S' is not external*

This error message indicates that the symbol referenced by ***#pragma extref*** cannot have an external reference emitted for it because the symbol is not external. An external reference can be emitted only for external functions which are not overloaded and for external data items.

760 *pre-compiled header file being recreated (debugging info may change)*

The compiler has detected that the module being compiled was used to create debugging information for use by other modules. In order to maintain correctness, the pre-compiled header file must be recreated along with the object file.

761 *octal escape sequence out of range; truncated*

This message indicates that the octal escape sequence produces an integer that cannot fit into the required character type.

Example:

```
char *p = "\406";
```

762 *binary operator '%s' missing right operand*

There is no expression to the right of the indicated binary operator.

763 *binary operator '%s' missing left operand*

There is no expression to the left of the indicated binary operator.

764 *expression contains extra operand(s)*

The expression contains operand(s) without an operator

765 *expression contains consecutive operand(s)*

More than one operand found in a row.

766 *unmatched right parenthesis ")"*

The expression contains a right parenthesis ")" without a matching left parenthesis.

767 *unmatched left parenthesis "("*

The expression contains a left parenthesis "(" without a matching right parenthesis.

768 *no expression between parentheses "()"*

There is a matching set of parenthesis "()" which do not contain an expression.

769 *expecting ':' operator in conditional expression*

A conditional expression exists without the ':' operator.

770 *expecting '?' operator in conditional expression*

A conditional expression exists without the '?' operator.

771 *expecting first operand in conditional expression*

A conditional expression exists without the first operand.

772 *expecting second operand in conditional expression*

A conditional expression exists without the second operand.

773 *expecting third operand in conditional expression*

A conditional expression exists without the third operand.

774 *expecting operand after unary operator '%s'*

A unary operator without being followed by an operand.

775 *'%s' unexpected in constant expression*

'%s' not allowed in constant expression

776 *assembler: '%s'*

A warning has been issued by the #pragma inline assembler.

777 *expecting 'id' after '::' but found '%s'*

The '::' operator has an invalid token following it.

Example:

```
#define fn( x ) ((x)+1)

struct S {
    int inc( int y ) {
        return ::fn( y );
    }
};
```

778 *only constructors can be declared 'explicit'*

Currently, only constructors can be declared with the ***explicit*** keyword.

Example:

```
int explicit fn( int x ) {
    return x + 1;
}
```

779 *const_cast type must be pointer, member pointer, or reference*

The type specified in a ***const_cast*** operator must be a pointer, a pointer to a member of a class, or a reference.

Example:

```
extern int const *p;
long lp = const_cast<long>( p );
```

780 *const_cast expression must be pointer to same kind of object*

Ignoring ***const*** and ***volatile*** qualification, the expression must be a pointer to the same type of object as that specified in the ***const_cast*** operator.

Example:

```
extern int const * ip;
long* lp = const_cast<long*>( ip );
```

781 *const_cast expression must be lvalue of the same kind of object*

Ignoring ***const*** and ***volatile*** qualification, the expression must be an lvalue or reference to the same type of object as that specified in the ***const_cast*** operator.

Example:

```
extern int const i;
long& lr = const_cast<long&>( i );
```

782 *expression must be pointer to member from same class in const_cast*

The expression must be a pointer to member from the same class as that specified in the ***const_cast*** operator.

Example:

```
struct B {
    int ib;
};
struct D : public B {
};
extern int const B::* imb;
int D::* imd const_cast<int D::*>( imb );
```

783 *expression must be member pointer to same type as specified in const_cast*

Ignoring ***const*** and ***volatile*** qualification, the expression must be a pointer to member of the same type as that specified in the ***const_cast*** operator.

Example:

```
struct B {
    int ib;
    long lb;
};
int D::* imd const_cast<int D::*>( &B::lb );
```

784 *reinterpret_cast expression must be pointer or integral object*

When a pointer type is specified in the ***reinterpret_cast*** operator, the expression must be a pointer or an integer.

Example:

```
extern float fval;
long* lp = const_cast<long*>( fval );
```

The expression has ***float*** type and so is illegal.

785 *reinterpret_cast expression cannot be casted to reference type*

When a reference type is specified in the ***reinterpret_cast*** operator, the expression must be an lvalue (or have reference type). Additionally, constness cannot be casted away.

Example:

```
extern long f;
extern const long f2;
long& lr1 = const_cast<long&>( f + 2 );
long& lr2 = const_cast<long&>( f2 );
```

Both initializations are illegal. The first cast expression is not an lvalue. The second cast expression attempts to cast away constness.

786 *reinterpret_cast expression cannot be casted to pointer to member*

When a pointer to member type is specified in the ***reinterpret_cast*** operator, the expression must be a pointer to member. Additionally, constness cannot be casted away.

Example:

```
extern long f;
struct S {
    const long f2;
    S();
};
long S::* mp1 = const_cast<long S:: *>( f );
long S::* mp2 = const_cast<long S:: *>( &S::f2 );
```

Both initializations are illegal. The first cast expression does not involve a member pointer. The second cast expression attempts to cast away constness.

787 *only integral arithmetic types can be used with reinterpret_cast*

Pointers can only be casted to sufficiently large integral types.

Example:

```
void* p;
float f = reinterpret_cast<float>( p );
```

The cast is illegal because ***float*** type is specified.

788 *only integral arithmetic types can be used with reinterpret_cast*

Only integral arithmetic types can be casted to pointer types.

Example:

```
float flt;
void* p = reinterpret_cast<void*>( flt );
```

The cast is illegal because `flt` has ***float*** type which is not integral.

789 *cannot cast away constness*

A cast or implicit conversion is illegal because a conversion to the target type would remove constness from a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S const * ps;
extern int const S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove constness.

790 *size of integral type in cast less than size of pointer*

An object of the indicated integral type is too small to contain the value of the indicated pointer.

Example:

```
int x;
char p = reinterpret_cast<char>( &x );
char q = (char)( &x );
```

Both casts are illegal since a ***char*** is smaller than a pointer.

791 *type cannot be used in reinterpret_cast*

The type specified with reinterpret_cast must be an integral type, a pointer type, a pointer to a member of a class, or a reference type.

Example:

```
void* p;
float f = reinterpret_cast<float>( p );
void* q = ( reinterpret_cast<void>( p ), p );
```

The casts specify illegal types.

792 *only pointers can be casted to integral types with reinterpret_cast*

The expression must be a pointer type.

Example:

```
void* p;
float f = reinterpret_cast<float>( p );
void* q = ( reinterpret_cast<void>( p ), p );
```

The casts specify illegal types.

793 *only integers and pointers can be casted to pointer types with reinterpret_cast*

The expression must be a pointer or integral type.

Example:

```
void* x;
void* p = reinterpret_cast<void*>( 16 );
void* q = ( reinterpret_cast<void*>( x ), p );
```

The casts specify illegal types.

794 *static_cast cannot convert the expression*

The indicated expression cannot be converted to the type specified with the ***static_cast*** operator. Perhaps reinterpret_cast or dynamic_cast should be used instead;

795 *static_cast cannot be used with the type specified*

A static cast cannot be used with a function type or array type.

Example:

```
typedef int fun( int );
extern int poo( long );
int i = ( static_cast<fun)( poo ) )( 22 );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

796 *static_cast cannot be used with the reference type specified*

The expression could not be converted to the specified type using static_cast.

Example:

```
long lng;
int& ref = static_cast<int&>( lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

797 *static_cast cannot be used with the pointer type specified*

The expression could not be converted to the specified type using static_cast.

Example:

```
long lng;
int* ref = static_cast<int*>( lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

798 *static_cast cannot be used with the member pointer type specified*

The expression could not be converted to the specified type using static_cast.

Example:

```
struct S {
    long lng;
};
int S::* mp = static_cast<int S::*>( &S::lng );
```

Perhaps reinterpret_cast or dynamic_cast should be used instead;

799 *static_cast type is ambiguous*

More than one constructor and/or used-defined conversion function can be used to convert the expression to the indicated type.

800 *cannot cast from ambiguous base class*

When more than one base class of a given type exists, with respect to a derived class, it is impossible to cast from the base class to the derived class.

Example:

```
struct Base { int b1; };
struct DerA public Base { int da; };
struct DerB public Base { int db; };
struct Derived public DerA, public DerB { int d; }
Derived* foo( Base* p )
{
    return static_cast<Derived*>( p );
}
```

The cast fails since `Base` is an ambiguous base class for `Derived.`

801 *cannot cast to ambiguous base class*

When more than one base class of a given type exists, with respect to a derived class, it is impossible to cast from the derived class to the base class.

Example:

```
struct Base { int b1; };
struct DerA public Base { int da; };
struct DerB public Base { int db; };
struct Derived public DerA, public DerB { int d; }
Base* foo( Derived* p )
{
    return (Base*)p;
}
```

The cast fails since `Base` is an ambiguous base class for `Derived.`

802 *can only static_cast integers to enumeration type*

When an enumeration type is specified with ***static_cast,*** the expression must be an integer.

Example:

```
enum sex { male, female };
sex father = static_cast<sex>( 1.0 );
```

The cast is illegal because the expression is not an integer.

803 *dynamic_cast cannot be used with the type specified*

A dynamic cast can only specify a reference to a class or a pointer to a class or ***void.*** When a class is referenced, it must have virtual functions defined within that class or a base class of that class.

804 *dynamic_cast cannot convert the expression*

The indicated expression cannot be converted to the type specified with the ***dynamic_cast*** operator. Only a pointer or reference to a class object can be converted. When a class object is referenced, it must have virtual functions defined within that class or a base class of that class.

805 *dynamic_cast requires class '%T' to have virtual functions*

The indicated class must have virtual functions defined within that class or a base class of that class.

806 *base class for type in dynamic_cast is ambiguous (will fail)*

The type in the ***dynamic_cast*** is a pointer or reference to an ambiguous base class.

Example:

```
struct A { virtual void f(){}; };
struct D1 : A { };
struct D2 : A { };
struct D : D1, D2 { };

A *foo( D *p ) {
    // will always return NULL
    return( dynamic_cast< A* >( p ) );
}
```

807 *base class for type in dynamic_cast is private (may fail)*

The type in the ***dynamic_cast*** is a pointer or reference to a private base class.

Example:

```
struct V { virtual void f(){}; };
struct A : private virtual V { };
struct D : public virtual V, A { };

V *foo( A *p ) {
    // returns NULL if 'p' points to an 'A'
    // returns non-NULL if 'p' points to a 'D'
    return( dynamic_cast< V* >( p ) );
}
```

808 *base class for type in dynamic_cast is protected (may fail)*

The type in the ***dynamic_cast*** is a pointer or reference to a protected base class.

Example:

```
struct V { virtual void f(){}; };
struct A : protected virtual V { };
struct D : public virtual V, A { };

V *foo( A *p ) {
    // returns NULL if 'p' points to an 'A'
    // returns non-NULL if 'p' points to a 'D'
    return( dynamic_cast< V* >( p ) );
}
```

809 *type cannot be used with an explicit cast*

The indicated type cannot be specified as the type of an explicit cast. For example, it is illegal to cast to an array or function type.

810 *cannot cast to an array type*

It is not permitted to cast to an array type.

Example:

```
typedef int array_type[5];
int array[5];
int* p = (array_type)array;
```

811 *cannot cast to a function type*

It is not permitted to cast to a function type.

Example:

```
typedef int fun_type( void );
void* p = (fun_type)0;
```

812 *implementation restriction: cannot generate RTTI info for '%T' (%d classes)*

The information for one class must fit into one segment. If the segment size is restricted to 64k, the compiler may not be able to emit the correct information properly if it requires more than 64k of memory to represent the class hierarchy.

813 *more than one default constructor for '%T'*

The compiler found more than one default constructor signature in the class definition. There must be only one constructor declared that accepts no arguments.

Example:

```
struct C {
    C();
    C( int = 0 );
};
C cv;
```

814 *user-defined conversion is ambiguous*

The compiler found more than one user-defined conversion which could be performed. The indicated functions that could be used are shown.

Example:

```
struct T {
    T( S const& );
};
struct S {
    operator T const& ();
};
extern S sv;
T const & tref = sv;
```

Either the constructor or the conversion function could be used; consequently, the conversion is ambiguous.

815 *range of possible values for type '%T' is %u to %u*

This informational message indicates the range of values possible for the indicated unsigned type.

Example:

```
unsigned char uc;
if( uc >= 0 );
```

Being unsigned, the char is always >= 0, so a warning will be issued. Following the warning, this informational message indicates the possible range of values for the unsigned type involved.

816 *range of possible values for type '%T' is %d to %d*

This informational message indicates the range of values possible for the indicated signed type.

Example:

```
signed char c;
if( c <= 127 );
```

Because the value of signed char is always <= 127, a warning will be issued. Following the warning, this informational message indicates the possible range of values for the signed type involved.

817 *constant expression in comparison has value %d*

This informational message indicates the value of the constant expression involved in a comparison which caused a warning to be issued.

Example:

```
unsigned char uc;
if( uc >= 0 );
```

Being unsigned, the char is always >= 0, so a warning will be issued. Following the warning, this informational message indicates the constant value (0 in this case) involved in the comparison.

818 *constant expression in comparison has value %u*

This informational message indicates the value of the constant expression involved in a comparison which caused a warning to be issued.

Example:

```
signed char c;
if( c <= 127 );
```

Because the value of char is always <= 127, a warning will be issued. Following the warning, this informational message indicates the constant value (127 in this case) involved in the comparison.

819 *conversion of const reference to non-const reference*

A reference to a constant object is being converted to a reference to a non-constant object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:

```
extern int const & const_ref;
int & non_const_ref = const_ref;
```

820 *conversion of volatile reference to non-volatile reference*

A reference to a volatile object is being converted to a reference to a non-volatile object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:

```
extern int volatile & volatile_ref;
int & non_volatile_ref = volatile_ref;
```

821 *conversion of const volatile reference to plain reference*

A reference to a constant and volatile object is being converted to a reference to a non-volatile and non-constant object. This can only be accomplished by using an explicit or `const_cast` cast.

Example:

```
extern int const volatile & const_volatile_ref;
int & non_const_volatile_ref = const_volatile_ref;
```

822 *current declaration has type '%T'*

This informational message indicates the type of the current declaration that caused the message to be issued.

Example:

```
extern int __near foo( int );
extern int __far foo( int );
```

823 *only a non-volatile const reference can be bound to temporary*

The expression being bound to a reference will need to be converted to a temporary of the type referenced. This means that the reference will be bound to that temporary and so the reference must be a non-volatile const reference.

Example:

```
extern int * pi;
void * & r1 = pi;               // error
void * const & r2 = pi;         // ok
void * volatile & r3 = pi;      // error
void * const volatile & r4 = pi;// error
```

824 *conversion of pointer to member across a virtual base*

In November 1995, the Draft Working Paper was amended to disallow pointer to member conversions when the source class is a virtual base of the target class. This situation is treated as a warning (unless -za is specified to require strict conformance), as a temporary measure. In the future, an error will be diagnosed for this situation.

Example:

```
struct B {
    int b;
};

struct D : virtual B {
    int d;
};
int B::* mp_b = &B::b;
int D::* mp_d = mp_b;       // conversion across a
virtual base
```

825 *declaration cannot be in the same scope as namespace '%S'*

A namespace name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the namespace.

Example:

```
namespace x {
    int q;
};
int x;
```

826 *'%S' cannot be in the same scope as a namespace*

A namespace name must be unique across the entire C++ program. Any other use of a name cannot be in the same scope as the namespace.

Example:

```
int x;
namespace x {
    int q;
};
```

827 *File: %s*

This informative message is written when the -ew switch is specified on a command line. It indicates the name of the file in which an error or warning was detected. The message precedes a group of one or more messages written for the file in question. Within each group, references within the file have the format `(line[,column])`.

828 *%s*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected either before or after the source file was read during the compilation process.

829 *%s: %s*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected while processing the switches specified in a command file or by the contents of an environment variable. The switch that was being processed is displayed following the name of the file or the environment variable.

830 *%s: %S*

This informative message is written when the -ew switch is specified on a command line. It indicates the location of an error when the error was detected while generating a function, such as a constructor, destructor, or assignment operator or while generating the machine instructions for a function which has been analysed. The name of the function is given following text indicating the context from which the message originated.

831 *possible override is '%S'*

The indicated function is ambiguous since that name was defined in more than one base class and one or more of these functions is virtual. Consequently, it cannot be decided which is the virtual function to be used in a class derived from these base classes.

832 *function being overridden is '%S'*

This informational message indicates a function which cannot be overridden by a virtual function which has ellipsis parameters.

833 *name does not reference a namespace*

A ***namespace*** alias definition must reference a ***namespace*** definition.

Example:

```
typedef int T;
namespace a = T;
```

834 *namespace alias cannot be changed*

A ***namespace*** alias definition cannot change which ***namespace*** it is referencing.

Example:

```
namespace ns1 { int x; }
namespace ns2 { int x; }
namespace a = ns1;
namespace a = ns2;
```

835 *cannot throw undefined class object*

C++ does not allow undefined classes to be copied and so an undefined class object cannot be specified in a ***throw*** expression.

836 *symbol has different type than previous symbol in same declaration*

This warning indicates that two symbols in the same declaration have different types. This may be intended but it is often due to a misunderstanding of the C++ declaration syntax.

Example:

```
// change to:
//   char *p;
//   char q;
// or:
//   char *p, *q;
char* p, q;
```

837 *companion definition is '%S'*

This informational message indicates the other symbol that shares a common base type in the same declaration.

838 *syntax error; default argument cannot be processed*

The default argument contains unbalanced braces or parenthesis. The default argument cannot be processed in this form.

839 *default argument started %L*

This informational message indicates where the default argument started so that any problems with missing braces or parenthesis can be fixed quickly and easily.

Example:

```
struct S {
    int f( int t= (4+(3-7), // missing parenthesis
    );
};
```

840 *'%N' cannot be declared in a namespace*

A ***namespace*** cannot contain declarations or definitions of ***operator new*** or ***operator delete*** since they will never be called implicitly in a ***new*** or ***delete*** expression.

Example:

```
namespace N {
    void *operator new( unsigned );
    void operator delete( void * );
};
```

841 *namespace cannot be defined in a non-namespace scope*

A ***namespace*** can only be defined in either the global namespace scope (file scope) or a namespace scope.

Example:

```
struct S {
    namespace N {
        int x;
    };
}
```

842 *namespace '::' qualifier cannot be used in this context*

Qualified identifiers in a class context are allowed for declaring ***friend*** functions. A ***namespace*** qualified name can only be declared in a namespace scope that encloses the qualified name's namespace.

Example:

```
namespace M {
    namespace N {
        void f();
        void g();
        namespace O {
            void N::f() {
              // error
            }
        }
    }
    void N::g() {
      // OK
    }
}
```

843 *cannot cast away volatility*

A cast or implicit conversion is illegal because a conversion to the target type would remove volatility from a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S volatile * ps;
extern int volatile S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove volatility.

844 *cannot cast away constness and volatility*

A cast or implicit conversion is illegal because a conversion to the target type would remove constness and volatility from a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S const volatile * ps;
extern int const volatile S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to remove constness and volatility.

845 *cannot cast away unaligned*

A cast or implicit conversion is illegal because a conversion to the target type would add alignment to a pointer, reference, or pointer to member.

Example:

```
struct S {
    int s;
};
extern S _unaligned * ps;
extern int _unaligned S::* mps;
S* ps1 = ps;
S& rs1 = *ps;
int S::* mp1 = mps;
```

The three initializations are illegal since they are attempts to add alignment.

846 *subscript expression must be integral*

Both of the operands of the indicated index expression are pointers. There may be a missing indirection or function call.

Example:

```
int f();
int *p;
int g() {
    return p[f];
}
```

847 *extension: non-standard user-defined conversion*

An extended conversion was allowed. The latest draft of the C++ working paper does not allow a user-defined conversion to be used in this context. As an extension, the WATCOM compiler supports the conversion since substantial legacy code would not compile without the extension.

848 *useless using directive ignored*

This warning indicates that for most purposes, the ***using namespace*** directive can be removed.

Example:

```
namespace A {
    using namespace A;  // useless
};
```

849 *base class virtual function has not been overridden*

This warning indicates that a virtual function name has been overridden but in an incomplete manner, namely, a virtual function signature has been omitted in the overriding class.

Example:

```
struct B {
    virtual void f() const;
};
struct D : B {
    virtual void f();
};
```

850 *virtual function is '%S'*

This message indicates which virtual function has not been overridden.

851 *macro '%s' defined %L*

This informational message indicates where the macro in question was defined. The message is displayed following an error or warning diagnostic for the macro in question.

Example:

```
#define mac(a,b,c) a+b+c

int i = mac(6,7,8,9,10);
```

The expansion of macro `mac` is erroneous because it contains too many arguments. The informational message will indicate where the macro was defined.

852 *expanding macro '%s' defined %L*

These informational messages indicate the macros that are currently being expanded, along with the location at which they were defined. The message(s) are displayed following a diagnostic which is issued during macro expansion.

853 *conversion to common class type is impossible*

The conversion to a common class is impossible. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type is impossible.

854 *conversion to common class type is ambiguous*

The conversion to a common class is ambiguous. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : public A { B(); };
class C : public A { C(); };
class D : public B, public C { D(); };
extern A a;
extern D d;
int i = ( a == d );
```

The last statement is erroneous since a conversion to a common class type is ambiguous.

855 *conversion to common class type requires private access*

The conversion to a common class violates the access permission which was private. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : private A { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type violates the (private) access permission.

856 *conversion to common class type requires protected access*

The conversion to a common class violates the access permission which was protected. One or more of the left and right operands are class types. The informational messages indicate these types.

Example:

```
class A { A(); };
class B : protected A { B(); };
extern A a;
extern B b;
int i = ( a == b );
```

The last statement is erroneous since a conversion to a common class type violates the (protected) access permission.

857 *namespace lookup is ambiguous*

A lookup for a name resulted in two or more non-function names being found. This is not allowed according to the C++ working paper.

Example:

```
namespace M {
    int i;
}
namespace N {
    int i;
    using namespace M;
}
void f() {
    using namespace N;
    i = 7;      // error
}
```

858 *ambiguous namespace symbol is '%S'*

This informational message shows a symbol that conflicted with another symbol during a lookup.

859 *attempt to static_cast from a private base class*

An attempt was made to static_cast a pointer or reference to a private base class to a derived class.

Example:

```
struct PrivateBase {
};

struct Derived : private PrivateBase {
};

extern PrivateBase* pb;
extern PrivateBase& rb;
Derived* pd = static_cast<Derived*>( pb );
Derived& rd = static_cast<Derived&>( rb );
```

The last two statements are erroneous since they would involve a ***static_cast*** from a private base class.

860 *attempt to static_cast from a protected base class*

An attempt was made to static_cast a pointer or reference to a protected base class to a derived class.

Example:

```
struct ProtectedBase {
};

struct Derived : protected ProtectedBase {
};

extern ProtectedBase* pb;
extern ProtectedBase& rb;
Derived* pd = static_cast<Derived*>( pb );
Derived& rd = static_cast<Derived&>( rb );
```

The last two statements are erroneous since they would involve a ***static_cast*** from a protected base class.

861 *qualified symbol cannot be defined in this scope*

This message indicates that the scope of the symbol is not nested in the current scope. This is a restriction in the C++ language.

Example:

```
namespace A {
    struct S {
        void ok();
        void bad();
    };
    void ok();
    void bad();
};
void A::S::ok() {
}
void A::ok() {
}
namespace B {
    void A::S::bad() {
        // error!
    }
    void A::bad() {
        // error!
    }
};
```

862 *using declaration references non-member*

This message indicates that the entity referenced by the ***using*** declaration is not a class member even though the ***using*** declaration is in class scope.

Example:

```
namespace B {
    int x;
};
struct D {
    using B::x;
};
```

863 *using declaration references class member*

This message indicates that the entity referenced by the ***using*** declaration is a class member even though the ***using*** declaration is not in class scope.

Example:

```
struct B {
    int m;
};
using B::m;
```

864 *invalid suffix for a constant*

An invalid suffix was coded for a constant.

Example:

```
__int64 a[] = {
    0i7, // error
    0i8,
    0i15, // error
    0i16,
    0i31, // error
    0i32,
    0i63, // error
    0i64,
};
```

865 *class in using declaration ('%T') must be a base class*

A ***using*** declaration declared in a class scope can only reference entities in a base class.

Example:

```
struct B {
    int f;
};
struct C {
    int g;
};
struct D : private C {
    B::f;
};
```

866 *name in using declaration is already in scope*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:

```
namespace B {
    int f;
    using B::f;
};
```

867 *conflict with a previous using-decl '%S'*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:

```
namespace B {
    int f;
    using B::f;
};
```

868 *conflict with current using-decl '%S'*

A ***using*** declaration can only reference entities in other scopes. It cannot reference entities within its own scope.

Example:

```
namespace B {
    int f;
    using B::f;
};
```

869 *use of '%N' requires build target to be multi-threaded*

The compiler has detected a use of a run-time function that will create a new thread but the current build target indicates only single-threaded C++ source code is expected. Depending on the user's environment, enabling multi-threaded applications can involve using the "-bm" option or selecting multi-threaded applications through a dialogue.

870 *implementation restriction: cannot use 64-bit value in switch statement*

The use of 64-bit values in switch statements has not been implemented.

871 *implementation restriction: cannot use 64-bit value in case statement*

The use of 64-bit values in case statements has not been implemented.

872 *implementation restriction: cannot use __int64 as bit-field base type*

The use of ***__int64*** for the base type of a bit-field has not been implemented.

873 *'based' function object cannot be placed in non-code segment "%s".*

Use ***__segname*** with the default code segment "_CODE", or a code segment with the appropriate suffix (indicated by informational message).

Example:

```
int __based(__segname("foo")) f() {return 1;}
```

Example:

```
int __based(__segname("_CODE")) f() {return 1;}
```

874 *Use a segment name ending in "%s", or the default code segment "_CODE".*

This informational message explains how to use ***__segname*** to name a code segment.

875 *RTTI must be enabled to use feature (use 'xr' option)*

RTTI must be enabled by specifying the 'xr' option when the compiler is invoked. The error message indicates that a feature such as ***dynamic_cast,*** or ***typeid*** has been used without enabling RTTI.

876 *'typeid' class type must be defined*

The compile-time type of the expression or type must be completely defined if it is a class type.

Example:

```
struct S;
void foo( S *p ) {
    typeid( *p );
    typeid( S );
}
```

877 *cast involves unrelated member pointers*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. These casts were illegal, but became legal when the new-style casts were added to the draft working paper.

Example:

```
struct C1 {
    int foo();
};
struct D1 {
    int poo();
};

typedef int (C1::* C1mp )();

C1mp fmp = (C1mp)&D1::poo;
```

The cast on the last line of the example would be diagnosed.

878 *unexpected type modifier found*

A ***__declspec*** modifier was found that could not be applied to an object or could not be used in this context.

Example:

```
__declspec(thread) struct S {
};
```

879 *invalid bit-field name '%N'*

A bit-field can only have a simple identifier as its name. A qualified name is also not allowed for a bit-field.

Example:

```
struct S {
    int operator + : 1;
};
```

880 *%u padding byte(s) added*

This warning indicates that some extra bytes have been added to a class in order to align member data to its natural alignment.

Example:

```
#pragma pack(push,8)
struct S {
    char c;
    double d;
};
#pragma pack(pop);
```

881 *cannot be called with a '%T *'*

This message indicates that the virtual function cannot be called with a pointer or reference to the current class.

882 *cast involves an undefined member pointer*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. In this case, the host class of at least one member pointer was not a fully defined class and, as such, it is unknown whether the host classes are related through derivation. These casts were illegal, but became legal when the new-style casts were added to the draft working paper.

Example:

```
struct C1 {
    int foo();
};
struct D1;

typedef int (C1::* C1mp )();
typedef int (D1::* D1mp )();

C1mp fn( D1mp x ) {
    return (C1mp) x;
}
// D1 may derive from C1
```

The cast on the last line of the example would be diagnosed.

883 *cast changes both member pointer object and class type*

This warning is issued to indicate that a dangerous cast of a member pointer has been used. This occurs when there is an explicit cast between sufficiently unrelated types of member pointers that the cast must be implemented using a reinterpret_cast. In this case, the host classes of the member pointers are related through derivation and the object type is also being changed. The cast can be broken up into two casts, one that changes the host class without changing the object type, and another that changes the object type without changing the host class.

Example:

```
struct C1 {
    int fn1();
};
struct D1 : C1 {
    int fn2();
};

typedef int (C1::* C1mp )();
typedef void (D1::* D1mp )();

C1mp fn( D1mp x ) {
    return (C1mp) x;
}
```

The cast on the last line of the example would be diagnosed.

884 *virtual function '%S' has a different calling convention*

This error indicates that the calling conventions specified in the virtual function prototypes are different. This means that virtual function calls will not function properly since the caller and callee may not agree on how parameters should be passed. Correct the problem by deciding on one calling convention and change the erroneous declaration.

Example:

```
struct B {
    virtual void __cdecl foo( int, int );
};
struct D : B {
    void foo( int, int );
};
```

885 *#endif matches #if in different source file*

This warning may indicate a ***#endif*** nesting problem since the traditional usage of ***#if*** directives is confined to the same source file. This warning may often come before an error and it is hoped will provide information to solve a preprocessing directive problem.

886 *preprocessing directive found %L*

This informational message indicates the location of a preprocessing directive associated with the error or warning message.

887 *unary '-' of unsigned operand produces unsigned result*

When a unary minus ('-') operator is applied to an unsigned operand, the result has an unsigned type rather than a signed type. This warning often occurs because of the misconception that '-' is part of a numeric token rather than as a unary operator. The work-around for the warning is to cast the unary minus operand to the appropriate signed type.

Example:

```
extern void u( int );
extern void u( unsigned );
void fn( unsigned x ) {
    u( -x );
    u( -2147483648 );
}
```

888 *trigraph expansion produced '%c'*

Trigraph expansion occurs at a very low-level so it can affect string literals that contain question marks. This warning can be disabled via the command line or ***#pragma warning*** directive.

Example:

```
// string expands to "(?]?~????"!
char *e = "(???)???-????";
// possible work-arounds
char *f = "(" "???" ")" "???" "-" "????";
char *g = "(\?\?\?)\?\?\?-\?\?\?\?";
```

889 *hexadecimal escape sequence out of range; truncated*

This message indicates that the hexadecimal escape sequence produces an integer that cannot fit into the required character type.

Example:

```
char *p = "\x0aCache Timings\x0a";
```

890 *undefined macro '%s' evaluates to 0*

The ISO C/C++ standard requires that undefined macros evaluate to zero during preprocessor expression evaluation. This default behaviour can often mask incorrectly spelled macro references. The warning is useful when used in critical environments where all macros will be defined.

Example:

```
#if _PRODUCTI0N // should be _PRODUCTION
#endif
```

891 *char constant has value %u (more than 8 bits)*

The ISO C/C++ standard requires that multi-char character constants be accepted with an implementation defined value. This default behaviour can often mask incorrectly specified character constants.

Example:

```
int x = '\0x1a'; // warning
int y = '\x1a';
```

892 *promotion of unadorned char type to int*

This message is enabled by the hidden -jw option. The warning may be used to locate all places where an unadorned char type (i.e., a type that is specified as ***char*** and neither ***signed char*** nor ***unsigned char***). This may cause portability problems since compilers have freedom to specify whether the unadorned char type is to be signed or unsigned. The promotion to ***int*** will have different values, depending on the choice being made.

893 *switch statement has no case labels*

The switch statement referenced in the warning did not have any case labels. Without case labels, a switch statement will always jump to the default case code.

Example:

```
void fn( int x )
{
    switch( x ) {
    default:
        ++x;
    }
}
```

D. Watcom C/C++ Run-Time Messages

The following is a list of error messages produced by the Watcom C/C++ run-time library. These messages can only appear during the execution of an application built with one of the C run-time libraries.

D.1 Run-Time Error Messages

Assertion failed: %s, file %s, line %d

This message is displayed whenever an assertion that you have made in your program is not true.

Stack Overflow!

Your program is trying to use more stack space than is available. If you believe that your program is correct, you can increase the size of the stack by using the "option stack=nnnn" when you link the program. The stack size can also be specified with the "k" option if you are using WCL or WCL386.

Floating-point support not loaded

You have called one of the printf functions with a format of "%e", "%f", or "%g", but have not passed a floating-point value. The compiler generates a reference to the variable "_fltused_" whenever you pass a floating-point value to a function. During the linking phase, the extra floating-point formatting routines will also be brought into your application when "_fltused_" is referenced. Otherwise, you only get the non floating-point formatting routines.

****** NULL assignment detected***

This message is displayed if any of the first 32 bytes of your program's data segment has been modified. The check is performed just before your program exits to the operating system. All this message means is that sometime during the execution of your program, this memory was modified.

To find the problem, you must link your application with debugging information and use Watcom Debugger to monitor its execution. First, run the application with Watcom Debugger until it completes. Examine the first 16 bytes of the data segment ("examine __nullarea") and press the space bar to see the next 16 bytes. Any values that are not equal to '01' have been modified. Reload the application, set watch points on the modified locations, and start execution. Watcom Debugger will stop when the specified location(s) change in value.

D.2 errno Values and Their Meanings

The following errors can be generated by the C run-time library. These error codes correspond to the error types defined in `ERRNO.H`.

ENOENT *No such file or directory*

The specified file or directory cannot be found.

E2BIG *Argument list too big*

The argument list passed to the `spawn..., exec...` or `system` functions requires more than 128 bytes, or the environment information exceeds 32K.

ENOEXEC *Exec format error*

The executable file has an invalid format.

EBADF *Bad file number*

The file handle is not a valid file handle value or it does not correspond to an open file.

ENOMEM *Not enough memory*

There was not enough memory available to perform the specified request.

EACCES *Permission denied*

You do not have the required (or correct) permissions to access a file.

EEXIST *File exists*

An attempt was made to create a file with the O_EXCL (exclusive) flag when the file already exists.

EXDEV *Cross-device link*

An attempt was made to rename a file to a different device.

EINVAL *Invalid argument*

An invalid value was specified for one of the arguments to a function.

ENFILE *File table overflow*

All the FILE structures are in use, so no more files can be opened.

EMFILE *Too many open files*

There are no more file handles available, so no more files can be opened. The maximum number of file handles available is controlled by the "FILES=" option in the "CONFIG.SYS" file.

ENOSPC *No space left on device*

No more space is left for writing on the device, which usually means that the disk is full.

EDOM *Argument too large*

An argument to a math function is not in the domain of the function.

ERANGE *Result too large*

The result of a math function could not be represented (too small, or too large).

EDEADLK *Resource deadlock would occur*

A resource deadlock would occur with regards to locked files.

D.3 Math Run-Time Error Messages

The following errors can be generated by the math functions in the C run-time library. These error codes correspond to the exception types defined in `MATH.H` and returned by the `matherr` function when a math error occurs.

DOMAIN *Domain error*

An argument to the function is outside the domain of the function.

OVERFLOW *Overflow range error*

The function result is too large.

PLOSS *Partial loss of significance*

A partial loss of significance occurred.

SING *Argument singularity*

An argument to the function has a bad value (e.g., log(0.0)).

TLOSS *Total loss of significance*

A total loss of significance occurred. An argument to a function was too large to produce a meaningful result.

UNDERFLOW *Underflow range error*

The result is too small to be represented.

#

-

3

8

<

\

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X