

Paradigm C++ Reference Manual



Version 5.0

Paradigm Systems

The authors of this software make no expressed or implied warranty of any kind with regard to this software and in no event will be liable for incidental or consequential damages arising from the use of this product. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of the licensing agreement.

The information in this document is subject to change without notice.

Copyright © 1999 Paradigm Systems. All rights reserved.

Paradigm C++™ is a trademark of Paradigm Systems. Other brand and product names are trademarks or registered trademarks of their respective holders.

Version 5.0

January 5, 2000

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Paradigm Systems.

Paradigm Systems
3301 Country Club Road
Suite 2214
Endwell, NY 13760
USA

(607)748-5966
(607)748-5968 (FAX)
Sales information: info@devtools.com
Technical support: support@devtools.com
Web: <http://www.devtools.com>
FTP: <ftp://ftp.devtools.com>

For prompt attention to your technical questions, contact our technical support team via the Internet at support@devtools.com. Please note that our 90 days of free technical support is only available to registered users of Paradigm C++. If you haven't yet done so, take this time to register your products under the Paradigm C++ Help menu or online at <http://www.devtools.com>.

Paradigm's SurvivalPak maintenance agreement will give you unlimited free technical support plus automatic product updates for an additional 12 months. Call (800) 537-5043 to purchase this protection today.

Table of Contents

Chapter 1 Getting started

Starting Paradigm C++.....	11
The Paradigm C++ menu system.....	12
The Paradigm C++ IDE SpeedBar	13
Using SpeedMenus.....	13
Using the Edit window.....	14
Creating a new file.....	14
Navigating your source files.....	15
Working beyond the Edit window	15
Working with projects.....	16
Creating an embedded application.....	17
Configuring the remote connection.....	19
Stand-alone debugging	19
Debugging with Paradigm C++	20
Debugger SpeedButtons	21
Customizing Paradigm C++.....	22
Configuring the Paradigm C++ editor	23
Syntax highlighting.....	24
Customizing the SpeedBars.....	25
Setting Paradigm C++ preferences	27
Saving your Paradigm C++ settings	27
Using help in Paradigm C++.....	28
Online help organization.....	28
Getting help in Paradigm C++	28
Getting context-sensitive help	29
Accessing and using contents screens	29
Using the index.....	29
Searching for keywords.....	29
Help SpeedMenus.....	30
Contacting Paradigm.....	30

Chapter 2 Managing projects

What is project management?	31
Project management tools.....	31
Using the Project Manager.....	32
Project Manager reference	33
Creating a project.....	34
Setting options with the New Target dialog box.....	34
Specifying the source node types	35
Opening existing projects	36
Adding nodes	36
Deleting source nodes	36
Adding files without relative path information.....	36
Editing source node attributes	37
Adding target nodes to your project.....	37
Deleting target nodes.....	38
Editing target attributes using TargetExpert.....	38
Moving nodes within a project.....	38
Copying nodes in a project.....	38

Converting project files into makefiles.....	39
Customizing the Project window	39
Grouping sets of files with Source Pools	40
Creating a Source Pool.....	40
Translators, viewers, and tools.....	41
Adding translators and viewers.....	41

Chapter 3 Project options

Setting project options	45
Using Style Sheets.....	45
Predefined Style Sheets	46
The default project options	46
Managing Style Sheets	46
Attaching Style Sheets to nodes	47
Sharing style sheets between projects.....	47
Project Description Language files.....	48
Setting local overrides	49
View project options	50
Compiling projects	50
Compiling part of a project.....	51
Fixing compile-time errors.....	52
Viewing errors	52
Fixing errors.....	52
Project options reference	52
16-bit compiler options.....	53
Calling convention.....	53
C option.....	53
Pascal.....	53
Register	53
Memory model.....	54
Assume SS equals DS.....	54
Automatic far data	54
Page alignment for far segments.....	55
Borland C++-compatible far data	55
Far data threshold	55
Far virtual tables	55
Fast huge pointers	56
Model.....	56
Put constant strings in code segments	57
Processor.....	57
16-bit instruction set	57
Data alignment.....	58
Segment names code.....	59
Code.....	59
Segment name data	60
Initialized data	60
Uninitialized data	60
Segment names far data	61
Far initialized data	61

Far uninitialized data	62	Identifier length.....	77
Far virtual tables.....	62	Language compliance	78
32-bit compiler options	62	Nested comments	78
Paradigm optimizing compiler	63	Debugging.....	79
Intel optimizing compiler	63	Browser reference information in OBJs	79
32-bit compiler options	63	Debug information in OBJs	79
Calling conventions	63	Line numbers	79
Processor	64	Out-of-line inline functions	80
Build attributes	65	Test stack overflow	80
C++ options	65	Precompiler headers	81
C++ compatibility	65	Cache precompiled header.....	81
'deep' virtual bases.....	65	Precompiled header name	81
Calling convention mangling compatibility	66	Precompiled headers	81
Disable constructor displacements	66	Stop precompiling after header file	82
Do not treat 'char' as distinct type.....	66	Directories options	82
Don't restrict scope 'for' loop expression		Source directories	82
variables	66	Include	82
Pass class values via reference to temporary.....	67	Library	82
Push 'this' first for Pascal member functions.....	67	Source	82
Treat 'far' classes as 'huge'	67	Specifying multiple directories	83
Virtual base pointers.....	67	File search algorithms	83
Vtable pointer follows data members.....	68	#include-file search algorithms.....	83
Exception handling/RTTI.....	68	Library file search algorithms	83
Enable exceptions	68	Output directories	84
Enable run-time type information	68	Intermediate	84
General.....	70	Final	84
Zero-length empty base classes.....	70	Guidelines for entering directory names.....	84
Member pointers.....	70	\$INHERIT and \$ENV()	85
Honor precision of member pointers.....	70	\$INHERIT	85
Member pointer representation	70	\$ENV().....	85
Templates.....	71	Librarian options.....	85
Templates instance generation	71	Case-sensitive library.....	85
Virtual tables.....	71	Create extended dictionary.....	86
Virtual tables linkage.....	71	Generate list file	86
Compiler options	72	Library page size	86
Defines.....	72	Purge comment records.....	86
Defining macros from the IDE.....	73	Linker options	86
Defining macros on the command line.....	73	16-bit linker.....	86
Code generation.....	73	Discard nonresident name table.....	86
Allocate enums as ints	73	Enable 32-bit processing.....	86
Duplicate strings merged.....	73	Inhibit optimizing far call to near	87
fastthis	74	Initialize segments	87
Register variables	74	Segment alignment	87
Unsigned characters	75	Transfer resident names to nonresident names	
Floating point.....	75	table	87
Correct Pentium FDIV flaw	75	16-bit optimizations	88
No floating point	76	Chain fixup	88
Fast floating point.....	76	Iterate data.....	89
Compiler output.....	76	Minimize resource alignment	89
Autodependency information.....	76	Minimize segment alignment.....	89
Generate COMDEFs	77	32-bit linker.....	89
Generate underscores	77	Allow import by ordinal	89
Source	77	Committed stack size (in hexadecimal).....	90

Committed heap size (in hexadecimal).....	90	Potential errors.....	101
File alignment (in hexadecimal).....	90	Stop after ... errors.....	101
Image base address (in hexadecimal).....	91	Stop after ... warnings.....	101
Image is based.....	91	Optimization options.....	101
Maximum linker errors.....	91	General settings.....	102
Object alignment (in hexadecimal).....	91	16- and 32-bit.....	102
Reserved heap size (in hexadecimal).....	91	Common subexpression.....	102
Reserved stack size (in hexadecimal).....	92	Induction variables.....	102
Use incremental linker.....	92	Inline intrinsic functions.....	103
Verbose.....	92	16-bit only.....	104
General.....	92	Assume no pointer aliasing.....	104
Case-sensitive exports and imports.....	92	Copy propagation.....	104
Case-sensitive link.....	93	Dead code elimination.....	105
Code pack size.....	93	Global register allocation.....	105
Default libraries.....	93	Invariant code motion.....	105
Include debug information.....	93	Jump optimization.....	105
Pack code segments.....	94	Loop optimization.....	106
Subsystem version (major.minor).....	94	Suppress redundant loads.....	106
Map file.....	94	32-bit.....	106
Include source line numbers.....	94	General optimization settings.....	108
Map file.....	95	Disable all optimizations.....	108
Off.....	95	Use selected optimizations.....	108
Segments.....	95	Optimize for size.....	108
Publics.....	96	Optimize for speed.....	108
Print mangled names in map file.....	96	Command-line only options.....	109
Warnings.....	96	Object search paths.....	109
32-bit warnings.....	96	16-bit command-line options.....	109
No stack" warning.....	96	Compile to .ASM, then assemble.....	109
Warn duplicate symbol in .LIB.....	96	Compile to .OBJ, no link.....	110
Make options.....	97	Specify assembler.....	110
Autodependencies.....	97	Specify executable file name.....	110
None.....	97	Pass option to linker.....	110
Use.....	97	Create a MAP file.....	110
Cache.....	97	Compiler .OBJ to filename.....	110
Cache and display.....	97	C++ compile.....	110
Break make on.....	97	Compile to assembler.....	111
Warnings.....	97	Specify assembler option.....	111
Errors.....	97	Undefine symbol.....	111
Fatal errors.....	98	Linker supported command-line options.....	111
New node path.....	98	Generate 8087 instructions.....	111
Message options.....	98	Compile to 16-bit real-mode .AXE.....	111
ANSI violations.....	98	Enable backward compatibility options.....	111
Display warnings.....	98	Link 16-bit real-mode .AXE.....	111
All.....	98	Extended memory swapping.....	111
Selected.....	99	Enable 24-bit extended addressing.....	112
None.....	99	32-bit command-line switches.....	112
General.....	99	Generate a multi-threaded target.....	112
User-defined warnings.....	99	Link using 32-bit Windows API.....	112
Inefficient C++ coding.....	99	Link 32-bit console application.....	112
Inefficient coding.....	100	Link 32-bit .DLL file.....	112
Obsolete C++.....	100	Link 32-bit .EXE file.....	112
Portability.....	100	Compiler command-line options.....	112
Potential C++ errors.....	100	Command-line options by function.....	118

Chapter 4 Browsing through your code

Using the browser	125
Starting the browser.....	125
Browser views	125
Browsing objects (class overview)	126
Browsing global symbols	126
Search.....	126
Browser SpeedMenu	126
Browsing symbols in your code	126
Symbol declaration window	127
Browsing references	127
Class inspection window	127
Browser filters and letter symbols	127
To view all instances of a type of symbol	128
To hide all instances of a type of symbol	128
To change several filter settings at once.....	128
Customizing the browser	128

Chapter 5 Using the integrated debugger

Types of bugs	129
Run-time errors.....	129
Logic errors	129
Planning a debugging strategy	130
Starting a debugging session.....	130
Compiling with debug information.....	130
Running your program in the IDE.....	131
Specifying program arguments	131
Controlling program execution.....	131
Running to the cursor location.....	132
The execution point	132
Finding the execution point.....	133
Stepping through code	133
Stepping into.....	133
Stepping over.....	134
Debugging member functions	135
Running to a breakpoint.....	135
Pausing a program.....	135
Terminating the program.....	135
Using breakpoints.....	136
Debugging with breakpoints.....	136
Setting breakpoints	136
Setting an unconditional breakpoint.....	136
Setting a conditional breakpoint.....	136
Setting other breakpoints.....	137
Setting breakpoints after program execution begins	137
Creating conditional breakpoints	137
Removing breakpoints	139
From an Edit window	139
From an Edit window or the Disassembly pane of the CPU window	139
From the Breakpoints window	139
Disabling and enabling breakpoints.....	140

Viewing and editing code at a breakpoint.....	140
Viewing code at a breakpoint	140
Editing code at a breakpoint	140
Resetting invalid breakpoints	141
Using breakpoint groups	141
Creating a breakpoint group	141
Disabling or enabling a breakpoint group.....	141
Using breakpoint option sets	141
Creating a breakpoint option set.....	141
Associating a breakpoint with an option set....	141
Changing breakpoint options	142
Changing the color of breakpoint lines	142
Using the Breakpoints window	142
About the Breakpoints window	143
Integrated debugger features	143
Add breakpoint	143
Other	143
Source breakpoint	144
Address breakpoint	144
Data watch breakpoint	144
C++ exception breakpoint.....	144
Breakpoint Condition/Action Options	145
Names	145
Conditions	145
Expr. True	145
Pass Count.....	146
Actions	146
Break.....	146
Stop Log.....	147
Start Log.....	147
Log Expr	147
Eval Expr	147
Log Message	147
Disabe Group	147
Enable Group	147
Add Conditions/Actions.....	147
Edit Breakpoint dialog box.....	148
Examining program data values	148
Modifying program data values	148
Understanding watch expressions	148
Using Watches window	149
Adding a watch.....	149
Add Watch dialog box.....	149
Formatting watch expressions	150
Changing watch properties.....	151
Edit Watch dialog box	152
Disabling and enabling watches.....	152
Deleting a watch.....	152
Dynamic updates.....	153
Inspecting data elements	153
Evaluating and modifying expressions	154
Evaluating expressions	154
Modifying the values of variables.....	155

CPU window	156
Resizing the CPU window panes.....	157
The Disassembly pane	157
The Disassembly pane SpeedMenu.....	157
Run to Current.....	158
Set PC to current.....	158
Toggle Breakpoint	158
Go to Address.....	158
Go to current PC.....	158
Follow jump into Disassembly pane	159
Follow address into Dump pane	159
Show previous address.....	159
Go to source.....	159
Memory Dump pane	159
The Dump pane SpeedMenu.....	160
Display as	160
Follow address into Disassembly pane.....	160
Follow address into Stack pane	160
Machine Stack pane	160
The Stack pane SpeedMenu.....	161
Go to top frame	161
Go to top of stack.....	161
Registers pane	161
The Registers pane SpeedMenu.....	161
Increment register.....	162
Decrement register.....	162
Zero register.....	162
Change register.....	162
Show old registers	162
Flags pane	162
The Flags pane SpeedMenu	163
Toggle flag.....	163
Viewing function calls	163
Navigating to function calls.....	164
Chapter 6 Paradigm C++ compiler	
Using the command-line compiler.....	165
Command-line compiler syntax.....	165
Default settings	165
Compiler configuration files.....	166
Compiler response files	166
Compiler-option precedence rules	166
Entering directories for command-line options ..	166
Using PLINK	167
PLINK command-line syntax.....	167
PLINK.CFG file	168
Linker response files.....	169
Using PLINK with PCC.EXE.....	169
Paradigm C++ tools overview	170
Running the command-line tools	170
Memory and MAKESWAP.EXE	170
The run-time manager and tools.....	171

Chapter 7 Using MAKE

MAKE basics	173
BUILTINS.MAK.....	174
Using TOUCH.EXE.....	174
MAKE options	175
Setting default MAKE options	176
Compatibility with Microsoft's NMAKE	176
Using makefiles.....	177
Symbolic targets	177
Rules for symbolic targets	177
Explicit and implicit rules	178
Explicit rule syntax	178
Single targets with multiple rules.....	179
Implicit rule syntax.....	179
Explicit rules with implicit commands	180
Command syntax.....	180
Command prefixes.....	180
Using @	181
Using -num and -	181
Using &.....	181
Command operators.....	181
Debugging with temporary files	181
Using MAKE macros	182
Defining MAKE macros	182
String substitutions in MAKE macros	183
Default MAKE macros	183
Modifying default MAKE macros	184
Using MAKE directives.....	184
.autodepend	185
!error	185
Error-checking controls.....	186
!if and other conditional directives	186
!include.....	187
!message.....	187
.path.ext.....	187
.precious	188
.suffixes.....	188
!undef	188
Using macros in directives.....	188
Null macros	189

Chapter 8 PLIB.EXE

PLIB basics.....	191
PLIB options	191
Using PLIB response files	193
PLIB operation list.....	193
PLIB examples.....	194

Chapter 9 Exception handling

C++ exception handling	197
Exception declarations	198
Throwing an exception.....	198
Handling an exception.....	199
Exception specifications	200

Sample output when 'a' is the input	202
Constructors and destructors	203
Setting exception handling options	203
Unhandled exceptions	203
C-based structured exceptions	204
Using C-based exceptions in C++	204
Handling C-based exceptions	205

Chapter 10 Using inline assembly

Inline assembly syntax and usage	207
Inline assembly references to data and functions	208
Inline assembly and register variables	208
Inline assembly, offsets, and size overrides	209
Using C structure members	209
Using jump instructions and labels	210
Compiling with inline assembly	210
Using the built-in assembler (PASM)	211
Opcodes	211
String instructions	212
Jump instructions	213
Assembly directives	213

Chapter 11 Header files summary

Using precompiled headers	216
Setting file names	217
Precompiled header file overview	217
Precompiled header limits	217
Precompiled header rules	218
Optimizing precompiled headers	218
alloc.h	219
assert.h	220
ctype.h	220
dos.h	221
embedded.h	221
errno.h	222
fcntl.h	222
float.h	223
generic.h	223
io.h	223
iomanip.h	224
limits.h	224
malloc.h	224
math.h	225
mem.h	226
memory.h	227
new.h	227
process.h	227
search.h	227
setjmp.h	227
share.h	228
signal.h	228
stdarg.h	228
stddef.h	229
stdio.h	229

stdiostr.h	230
stdlib.h	230
string.h	231
sys\locking.h	231
sys\types.h	231
time.h	232
values.h	232
varargs.h	233
except.h	233
_defs.h	233
_nfile.h	233
_null.h	234

Chapter 12 Math

Floating-point I/O	235
Floating-point options	235
Emulating the 80x87 chip	235
Using the 80x87 code	236
No floating-point code	236
Fast floating-point option	236
The 87 environment variable	236
Registers and the 80x87	237
Disabling floating-point exceptions	237
Using complex types	238
Using bcd types	238
Converting bcd numbers	239
Number of decimal digits	240

Chapter 13 16-bit memory management

Running out of memory	241
Memory models	241
The 8086 registers	241
General-purpose registers	242
Segment registers	243
Special-purpose registers	243
The flags register	243
Memory segmentation	244
Address calculation	245
Pointers	245
Near pointers	246
Far pointers	246
Huge pointers	246
The five memory models	247
Mixed-model programming: Addressing modifiers	252
Segment pointers	252
Declaring far objects	253
Declaring functions to be near or far	253
Declaring pointers to be near, far, or huge	254
Pointing to a given segment:offset address	255
Using library files	255
Linking mixed modules	255

Chapter 14 Using iostreams classes

What is a stream?	257
-------------------------	-----

The iostream library.....	257	Fatal errors	267
The streambuf class	257	Errors	268
The ios class	258	Warnings	268
Stream output.....	259	Informational messages.....	268
Fundamental types	260	Message generators	268
I/O formatting.....	260	Compiler errors and warnings.....	268
Manipulators.....	260	Run-time errors and warnings.....	269
Filling and padding.....	261	Linker errors and warnings	269
Stream input.....	262	Paradigm C++ debugger messages	269
I/O of user-defined types.....	263	ObjectScripting error messages.....	270
Simple file I/O.....	263	Message formats	270
String stream processing.....	264	Symbols in messages.....	270
Appendix A Paradigm C++ errors and		Alphabetical list of messages	271
messages		Index.....	279
Message categories	267		

Getting started

Command-line diehards need not despair, a complete set of command tools and make utility is included with Paradigm C++.

Welcome to Paradigm C++, a state-of-the-art integrated development environment (IDE) for creating x86 embedded system applications in C, C++, and assembly language. With the Paradigm C++ IDE, you can create, debug, and deploy real-time embedded system applications without resorting to the use of external tools. If you are used to running separate editor, debugger, make, and other tools to get a job done, then you are in for a real treat with Paradigm C++.

To help you get familiar with all the powerful capabilities of the Paradigm C++ IDE, this guide offers an overview of the key technologies that work for you in Paradigm C++:

- Starting Paradigm C++
- Using SpeedMenus
- Using the Edit Window
- Working with projects
- Configuring the remote connection
- Debugging with Paradigm C++
- Customizing Paradigm C++
- Using help in Paradigm C++

At first, Paradigm C++ may take some getting used to since it breaks the old-style embedded system development metaphor of separate edit, compile, and debug tools, and instead tracks the way modern applications are generated. Paradigm C++ includes many powerful features you may not be familiar with, so it pays to explore its full potential before you jump headfirst into a new project. Take a look at the material we provide here and use it as the basis for creating and modifying your own projects.

Starting Paradigm C++

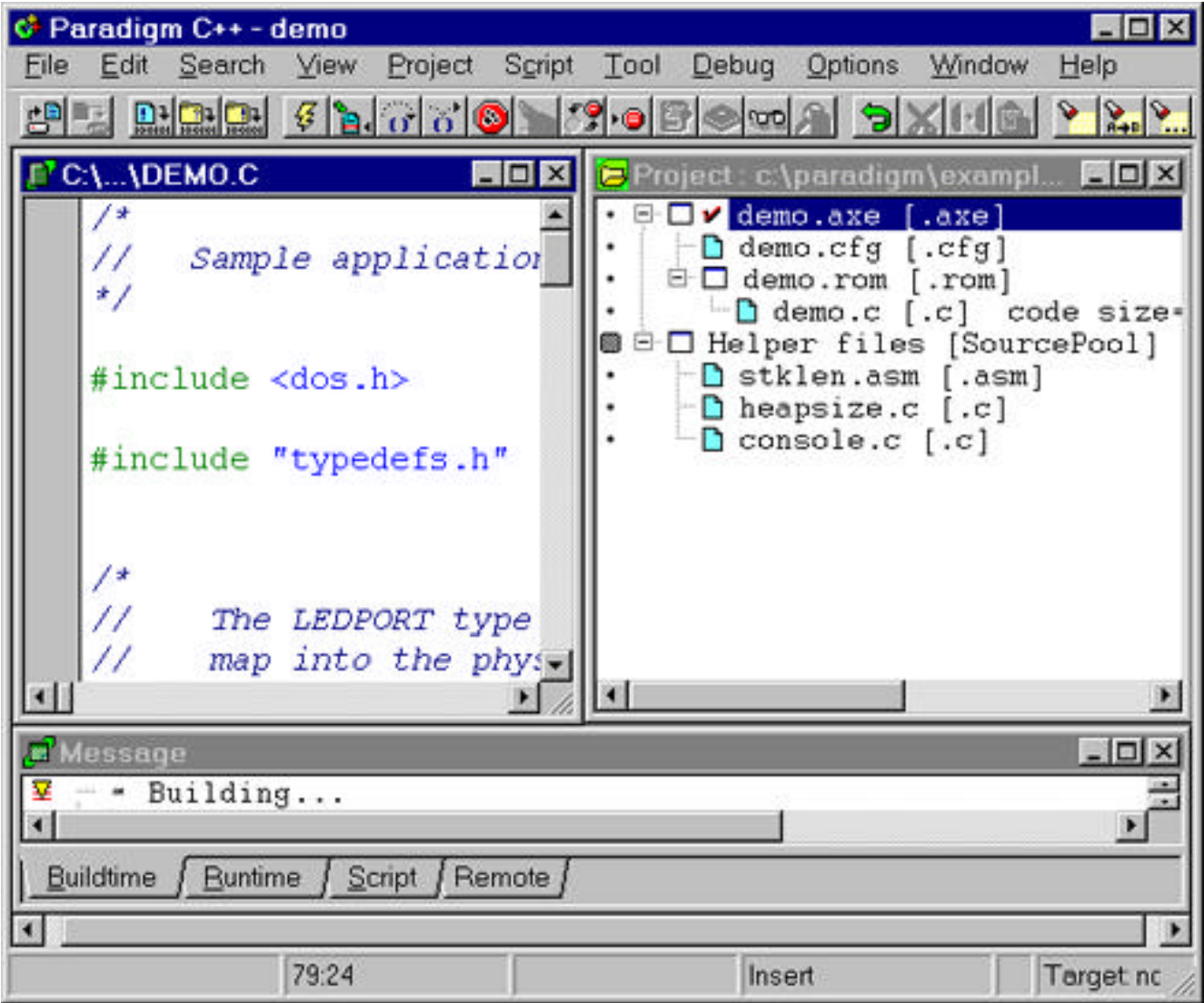
Or select INSTALL.EXE from the CD-ROM drive.

Installation instructions launch automatically from the Paradigm C++ CD. After following the instructions, exit the install screen. The Start menu will contain a program item titled Paradigm C++. Use the program item to launch Paradigm C++.

For optimum performance, Paradigm C++ requires a Pentium 120, Windows 95/NT, and 50MB hard disk space.

Figure 1-1, page 1-12 shows how Paradigm C++ looks after loading the DEMO.IDE project, opening DEMO.C, and building the application. The key features to note are the Menu Bar offering access to the various Paradigm C++ tools, the SpeedBar displaying context-sensitive shortcuts to relevant operations such as debugging and browsing, and the Status Bar at the very bottom with contains up-to-the minute information of the status of Paradigm C++. Filling the remainder of the window are the Edit, Project, Message and other views, where the real work of developing an embedded application will take place.

Figure 1-1 Paradigm C++ IDE screenshot



The Paradigm C++ menu system

The following table describes the menu options on the Paradigm C++ Menu Bar.

Table 1-1
Paradigm C++
global menus

Menu item	Command descriptions
File	Commands to open, save, and print files. Also includes the Paradigm C++ exit command along with a list of recently accessed files.
Edit	Clipboard command and commands for undoing and redoing operations on edit buffers.
Search	Commands for searching and replacing in edit buffers, files, or the current project, browsing symbols, locating functions, and reviewing error messages generated by the programming tools.
View	Commands to open the Project Manager, Message window, and Browser. Also contains commands to open the integrated debugger views during a debugging session.
Project	Commands to open, close, and build or make a project.
Script	Provides commands to run and test scripts to automate Paradigm C++. cScript is a powerful Paradigm C++ feature that allows you to automate and integrate tools into Paradigm C++.
Tool	Commands to launch any external programming tools from Paradigm C++.

Debug	Commands to run your project under control of the Paradigm C++ integrated debugger.
SCCS	Source code control system integration commands. This is an optional menu that is present when a source code control add-in is installed.
Options	Paradigm C++ customization and project configuration commands. Here is where you can completely tailor Paradigm C++ to work as you do.
Window	Paradigm C++ window management commands give you complete control to navigate between windows and close or minimize selected windows.
Help	Commands to access the Paradigm C++ online help are included here. Paradigm C++ includes extensive online help covering all of Paradigm C++, from the IDE operation to the details of the compiler run-time libraries.

More information about using cScript is available in the online Help.

Because Paradigm C++ is fully extensible by the end-user, there may be other entries on the menu bar from version control tools, real-time operating systems, and other third-party tools. With just a single line of Paradigm Scripting Language (cScript) code, you can have your favorite commands displayed here to use whenever you need them.

The Paradigm C++ IDE SpeedBar

The SpeedBar (located under the main menu) has buttons that give quick access to menu commands that relate to the area of Paradigm C++ you're working in. For example, if you're editing code, the SpeedBar contains cut and paste commands, file save commands, and so on, as well as commands to build and debug. When the Project window has focus, the SpeedBar has buttons that pertain to projects, such as commands for adding project nodes and browsing option settings.

Figure 1-2 Paradigm C++ IDE SpeedBar example



The Status Bar at the bottom of Paradigm C++ contains "flyby" help hints; when the cursor is over a button, the Status Bar describes the button command. You can configure the flyby hints and other SpeedBar options as described in "Customizing the SpeedBars," page 25. See Figure 1-7, page 1-22 for a description of the above Paradigm C++ SpeedButtons available during a debug session.

Using SpeedMenus

Right-clicking (clicking the right mouse button) accesses the Paradigm C++ *SpeedMenus*. SpeedMenus contain commands that are context-sensitive to the area of the program you're working in. For example, the SpeedMenu for the Edit window contains commands that are related to the editor. In the Project Manager, the SpeedMenus contain commands to help you with managing your project.

To get a feel for SpeedMenus, try the following:

If you installed Paradigm C++ in a different directory, adjust the paths used in this guide.

1. From the Paradigm C++ Menu Bar, choose Project|Open project, then select the project file DEMO.IDE in the PARADIGM\EXAMPLES\REAL\DEMO directory.
2. Double-click the DEMO.C node in the Project window to load the file in an Edit window so changes can be made.
3. Move the cursor to the **dos.h** header file reference by clicking on the file name in the source code.

4. Right-click to open the Edit window SpeedMenu, then choose Open Source to open an Edit window that contains this header file. You can do this even quicker using the by right-clicking anywhere in the DEMO.C Edit window and selecting the Include command. Paradigm C++ will instantly parse the file and extract all include file references in the buffer. Just select the desired include file and you are instantly there to begin making changes.



In addition to right-clicking, Paradigm C++ SpeedMenus can be accessed at any time by pressing *Alt-F10*.

Using the Edit window

Edit windows contain the Paradigm C++ editor, which you can use to create and edit your program code. When you're editing a file, the Paradigm C++ status bar displays the following information about the file that you are editing:

- The line number and character position of the cursor. For example, if the cursor is on the first line and first character of an Edit window, you'll see 1:1 in the Status Bar. If the cursor is on line 68 and character 23, you'll see 68:23.
- The edit mode: insert or overwrite. Press *Insert* to toggle whether your text additions overwrite existing characters or insert new ones into the file.
- The file's save status. The word Modified appears if you've made changes to the file in the active Edit window, and you have not yet saved your edits or changes.



The Paradigm C++ editor contains many powerful features to help you enter and modify your program code. For example, you can undo multiple edits by choosing Edit|Undo or pressing *Alt-Backspace*. You can also open multiple Edit windows; tile the windows as you wish; subdivide the window into different Edit panes; and cut, copy, and paste text between any open files. Paradigm C++ is supplied with four editor emulations and if these don't suffice, you can create your own editor from any of the supplied editors.

Although this chapter provides a brief introduction to the editor, complete details on how to use and customize the editor can be found in the online Help. Choose Help|Contents and double-click Paradigm C++ User's Guide. The Editor is discussed within the Integrated Development Environment (IDE) book topics.

Creating a new file

To introduce you to the editor, step through the following instructions to add a new source file to a sample embedded application.

1. If not already open, File|Open the DEMO.IDE project in PARADIGM\EXAMPLES\REAL\DEMO.
2. From the Paradigm C++ Menu Bar, choose File|New|Text Edit to open a new Edit window with an empty file.

By default, Paradigm C++ names new files NONAME xx .CPP, where xx is a number that is incremented with each new file opened. Don't worry about the filename for now, you'll be prompted to change it when you save the file.

3. In the Edit window, type the following C++ code to create a simple embedded program.

```

#include <stdio.h>

char buffer[128] ;

void main(void)
{
    unsigned passcount = 0 ;

    char* format = "%05u Welcome to ParadigmC++!\n" ;
    for (;;) {
        sprintf( buffer, format, passcount) ;
        passcount++ ;
    }
}

```

4. Choose File|Save, and save your new file with the file name TEST.C.

Although we created the file, it is not yet part of our Paradigm C++ project. Later, in “Creating an embedded application,” page 17, we will show you how to add this file to the project where it will get built with other source files in the project.

Navigating your source files

Once you have some text in the Edit window, you can navigate around your source code. Paradigm C++ utilizes instant-parsing technology to scan the current Edit window and extract information about functions, structures and classes, enumerations, and include files. In small files, source code navigation is possible by scrolling the Edit window, in large files and multi-file projects, it really isn't possible.

To really see the parsing technology in action, try the following test. Using the file TEST.C that was just created, right-click in the window and select the Functions - only 'main()' should appear at this time.

Now add a new function to the file, such as

```

int test(int x, int y)
{
    return x + y ;
}

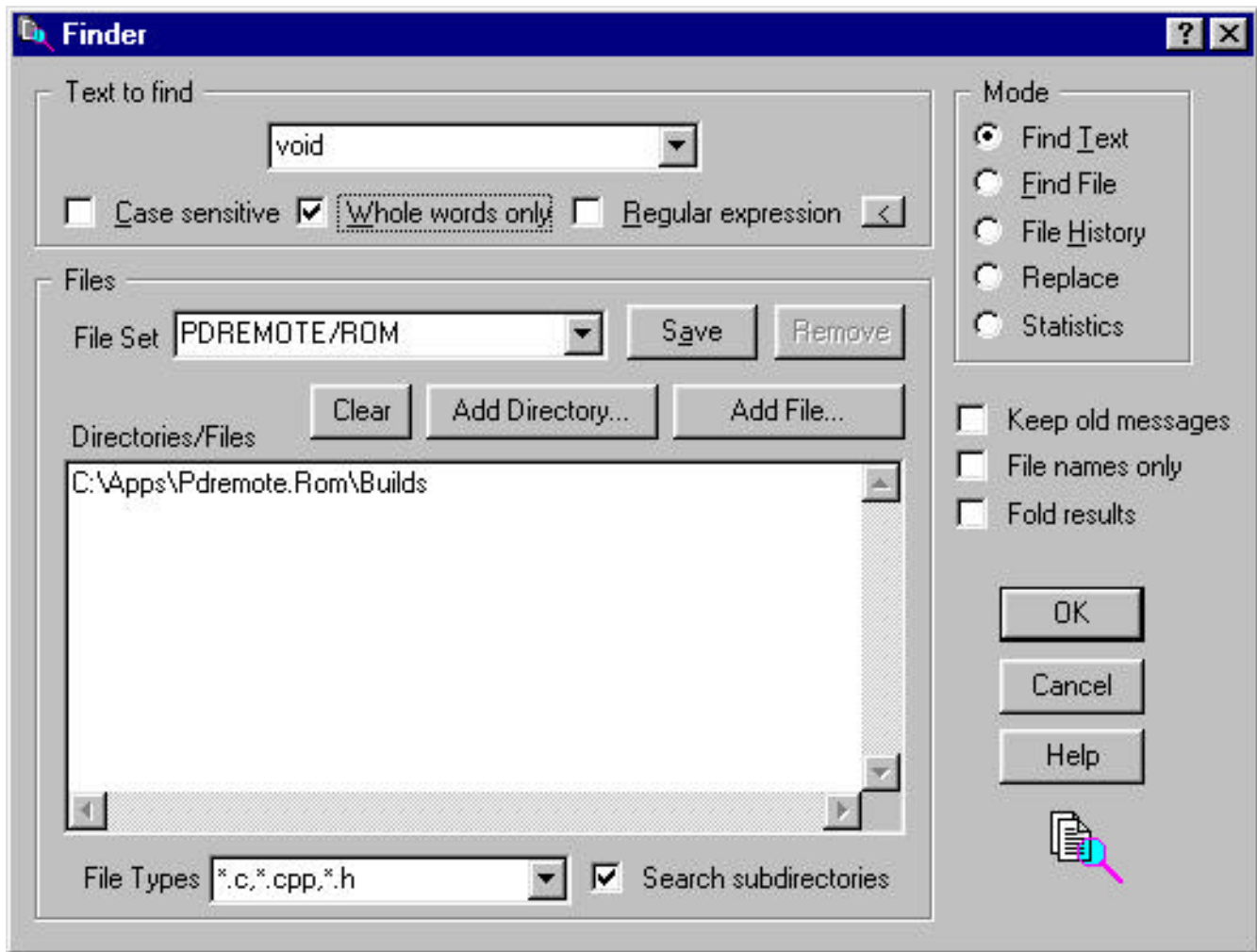
```

Now right-click in the window and select the Functions again and see that both main() and test() are in the list of functions in the file. No compiling, just instant access to your source code definitions to make it easy to navigate to any function, class or include file in the current Edit window.

Working beyond the Edit window

The Paradigm C++ Finder, found under the Search menu, can do much more to help with the software development process. The Finder provides the ability to search within a file, a project, or the entire disk drive for any regular expression. This is an incredibly powerful capability when you need to find text or make changes across one or many files in your project or on your disk.

Figure 1-3 Paradigm C++ Finder



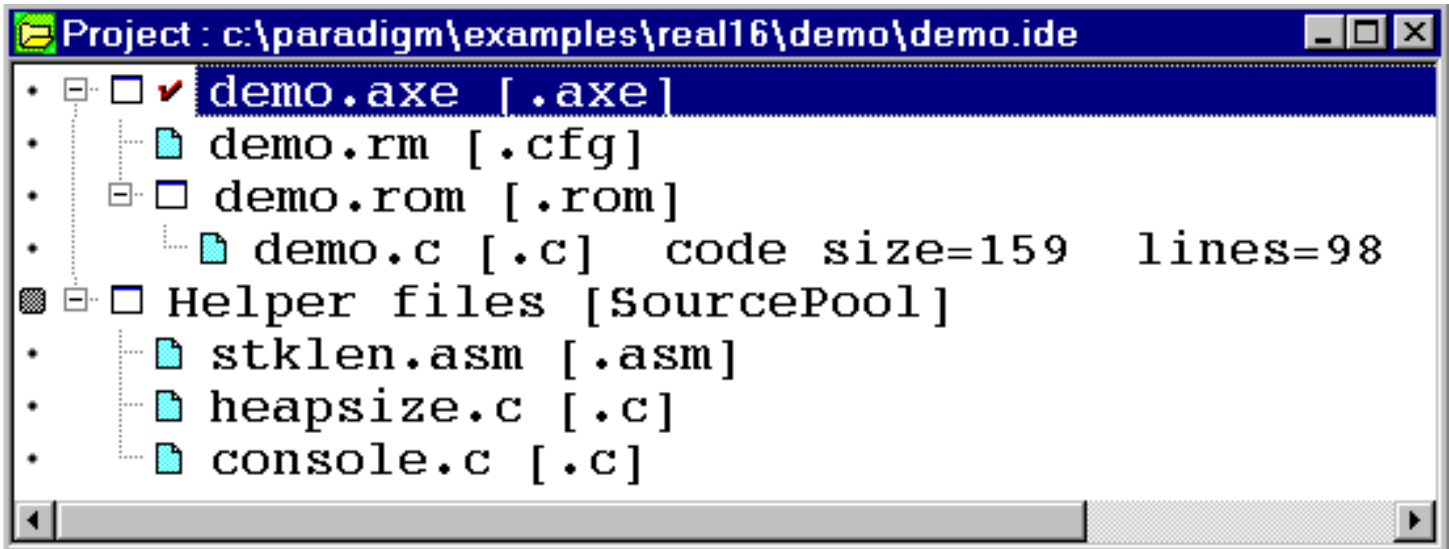
No matter what your needs, Paradigm C++ has the tools you need to manage and maintain your project files. While the Finder works on any source file, the browser adds even more power by using compiled code to create a database that can be utilized to find where a function is defined and all the instances that it is used in. More information about the browser is available in the online Help index under "browser" or see Chapter 4, "Browsing through your code," page 4-125 of this manual.

Working with projects

After you install Paradigm C++, you'll want to make sure the program is correctly set up; the details of the compiler and the Paradigm C++ IDE can wait until later. The best way to test your setup is to compile, build and load the sample applications included with Paradigm C++.

Paradigm C++ uses *projects* to help manage your code and make sure any source code changes are reflected in the other files that depend on them. As an application grows in size and complexity, it becomes dependent on various intermediate files. Often, source files need to be compiled with different compilers and different sets of compiler options. Even a simple embedded application can have multiple C/C++ source files, with each file type requiring different compilers and different compiler settings.

Figure 1-4 The Project Manager Project window



As your project complexity increases, the need increases for a way to manage the different components in the project. Looking at the files that make up a project, you can see that a project combines one or more source files to produce a single target file. While target files are usually a .AXE or .HEX file, source files cover a broader range of file types, including .C, .CPP, .ASM, and other files. Additionally, many source files have autodependent files (files that are automatically included by the source), such as C header files. In larger projects, you are likely to find several targets with scores of sources.

To get the most from Paradigm C++, we need to create a project so the files and build options are saved, just as they would be in a more traditional makefile.

Creating an embedded application

You can become familiar with the Project Manager and the C/C++ compiler by following these steps to create a simple embedded application:

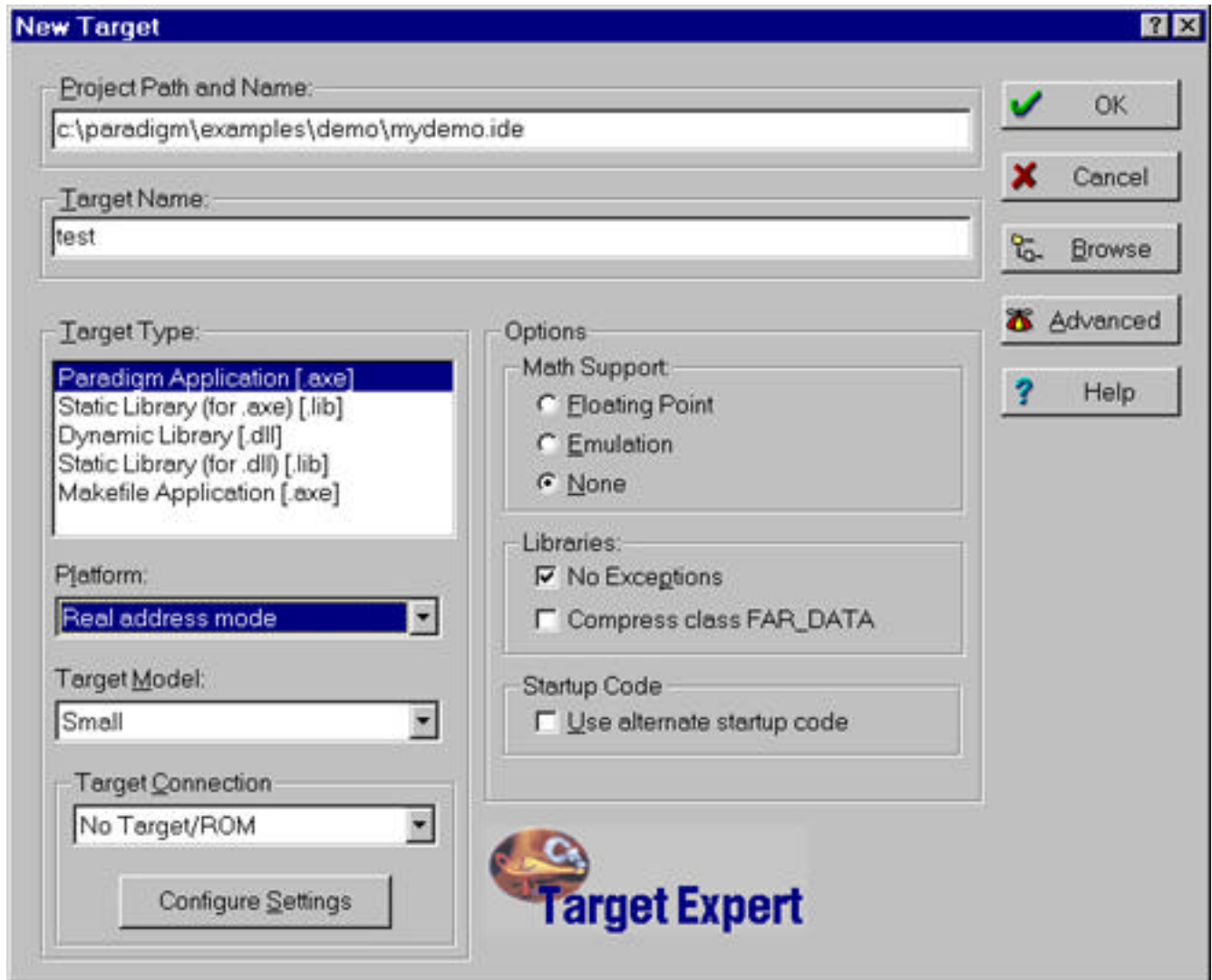
If the directory doesn't exist, Paradigm C++ creates the directory for you.

1. From the Paradigm C++ Menu Bar, choose File|New|Project..., then set the following options in the New Target dialog box:
 - Type the path and name for your new project in the Project Path and Name input box. In this case, type:
`\paradigm\examples\demo\mydemo.ide`
 - Type the target name you want to use. Because you can have more than one target in the project, you can have different names for targets that share files and options. In this case, type:
`test`
 - In the Target Type list box, click Paradigm Application [.AXE]. This selection will create a project where the source modules are compiled, assembled, linked and located to generate .AXE files for debugging or .HEX and .BIN files for placing within flash or EPROM devices.
 - If you like, select the desired platform and memory model you want for your application. You can also enable the use of floating point arithmetic or other options that depend on the selected Target Type.

- If you like, choose a target connection from the list of available remote connection interfaces.

The New Target dialog box should now resemble the one shown in Figure 1-5.

Figure 1-5 New Target dialog box



2. Choose *OK* to close the New Target dialog box.
3. The Project window opens and displays the target and dependencies of the project you just created.

The following are definitions for the nodes within this newly created project:

- TEST.AXE** This node is the final target node that is generated during the locate phase of the build and includes the absolute code for debugging or burning into flash or EPROM.
- TEST.CFG** This is the Paradigm LOCATE configuration file that is used to generate the TEST.AXE output file. This file contains the description of the target system address space as well as the build instructions for placing the program code and data at addresses that you specify.

TEST.ROM This node is generated by the Paradigm C++ linker and is also the point in the build process in which a .MAP file is generated for use in the locate phase.

TEST.C This node references the files TEST.C, the file that you created earlier in the chapter (if you haven't already done so, create this file by following the instructions listed in the section, "Creating a new file" on page 14).

4. Build the application by selecting the .AXE node and right-click to bring up the local options and select 'Build node'. Because Paradigm C++ has a built-in Project Manager, it always knows when the project is out-of date and needs to be rebuilt so there is no need to explicitly do this. We could have also selected the Project|Make all or the Project|Build all commands from the SpeedBar or from the Project menu.

If you correctly followed all the steps in this section, the application builds without errors. If the compiler reports errors or warnings during the compile, retrace the steps in this section to ensure you correctly followed the steps. When the program compiles without errors, the Project Manager creates an executable program called TEST.AXE and places it in the directory you selected when the project was created.

You can access the Master Index from within any online Help topic by right-clicking.

This is only a small fraction of the information on working with projects. See "projects" in the online Help index for complete details on managing the build process using Paradigm C++ projects or see Chapter 2 "Managing projects," on page 2-31 of this manual.

Configuring the remote connection

A target connection must be specified to begin debugging.

Paradigm C++ can only debug when a target such as PDREMOTE/ROM or an in-circuit emulator is connected. Configuring the remote connection gives Paradigm C++ the information it needs about your target to begin a debugging session. To configure the remote connection:

1. Select the TEST.AXE node from the project view of the Project Manager.
2. Right-click to see local menu options for the TEST.AXE node.
3. Select TargetExpert. The dialog contains a pull down menu for Target Connection.
4. Select the drop down menu to see a list of available remote connection interfaces and select the desired remote connection interface.
5. Press the Modify connection settings button to make specific changes to the remote interface settings.

Once the remote connection settings are set up, click *OK* to close the remote connection dialog. You are now ready to start debugging. Double-click the TEST.AXE node to rebuild the application (if needed) and download the application to the target.



The debugger sets a software breakpoint at the label **main** in the application. If you would rather start at the reset vector or run to a different place in the application, then select Options|Environment|Debugger|Debugger Behavior and delete **main**, or add the function name, to Run to on startup.

Stand-alone debugging

To configure the remote connection to do stand-alone debugging without the use of Project Manager,

1. Close any demo projects that may be open (Project|Close project).
2. Select Debug|Load, and *Browse* or type in the name of the .AXE (or .HEX) file for remote download, for example,
`\paradigm\examples\demo\test.axe`
3. Choose the desired remote connection interface.
4. Press the Modify settings button to change any specific remote connection settings for the selected interface.
5. Select OK to load the application file and start debugging without the use of the Project Manager.

Again, the debugger sets a software breakpoint at the label **main** in the application. If you would rather start at the reset vector or run to a different place in the application, then select Options|Environment|Debugger|Debugger Behavior and delete **main**, or add the function name, to Run to on startup.

6. When you would like to exit stand-alone debugging mode, Select Debug|Terminate debug session or hit *Ctrl-F2*.

Debugging with Paradigm C++

If you have multiple targets, you can select the target connector from the local menu of a target node in the Project view.

For a demonstration of debugging in Paradigm C++, open the DEMO.IDE project in PARADIGM\EXAMPLES\REAL\DEMO as you did in “Creating a new file,” page 14 and double-click the DEMO.C node in the Project window. Right-click the DEMO.AXE node in the Project window and select TargetExpert to ensure that the Target Connection is set to the desired remote connection. Then simply double-click the DEMO.AXE node in the Project window to download the application to your target. If the debugger option to execute to main(), found under Options|Environment|Debugger|Debugger Behavior, is enabled, the debugging session will look like Figure 1-6, page 1-21.

At this point the Debug menu commands and SpeedBar will come alive so you can inspect program data, view the processor registers, or access target peripherals. Right-clicking in the Edit window will bring up the debugger SpeedMenu for quick access to debugging commands.



Step over/into

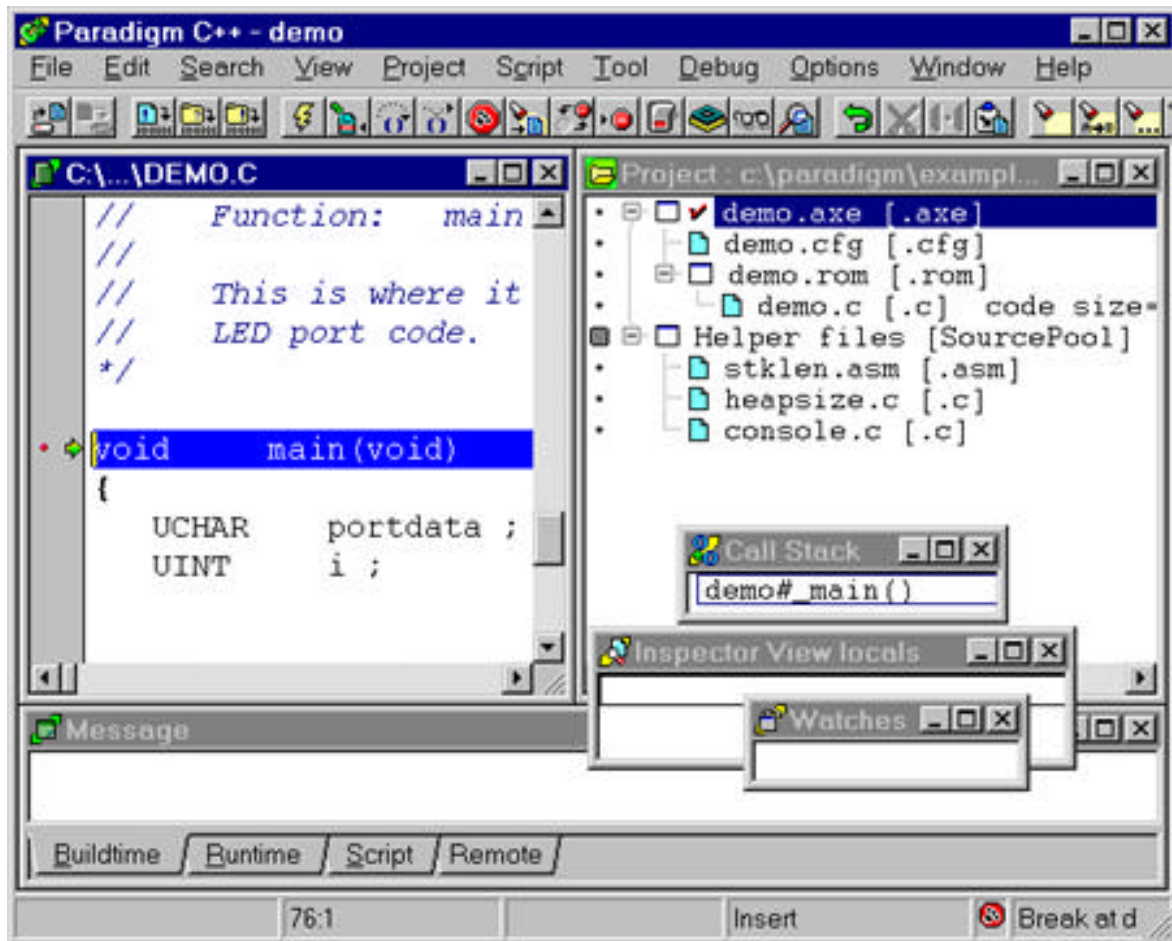


Run/Run to

You can step through the program and test until you find a bug that needs fixing. Use the Statement step into or Statement step over SpeedButtons located beneath the Menu bar to begin debugging. You could use the Run SpeedButton but the application won't stop unless you have set a breakpoint somewhere in the program. You can also use the Run to here button to execute to a particular source line that the cursor is on. See Figure 1-7, page 1-22 for a description of Paradigm C++ SpeedButtons available during a debug session.

When you find a problem, you might notice that there is no difference between editor windows and debugger windows. This is a big improvement over traditional tools since you can fix a bug right away without exiting the debugger. If you make a change, you can either continue the debugging session or you can rebuild the application and test the change - all without losing your place! This is where Paradigm C++ excels at making the most of your development time.

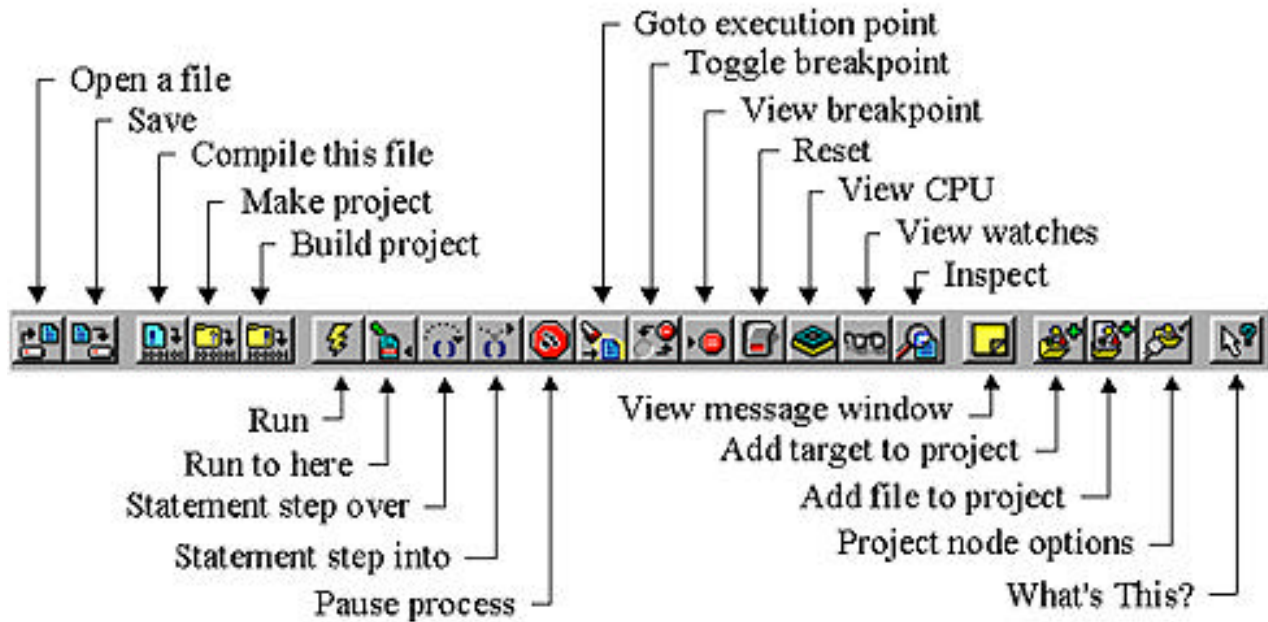
Figure 1-6 Paradigm C++ debugging session



Debugger SpeedButtons

This section will familiarize you with the Paradigm C++ SpeedButtons used in a debugging session.

Figure 1-7 Paradigm C++ SpeedButtons



In Paradigm C++, click the What's This? SpeedButton and then a SpeedButton of interest to receive a help description of that button. The Paradigm C++ debugger is covered in complete detail in Chapter 5 "Using the integrated debugger," page 5-129 of this manual. We have covered just the basics here. Plan on spending some time in Chapter 5 or see "integrated debugger" in the online Help index for more assistance on the Paradigm C++ integrated debugger.

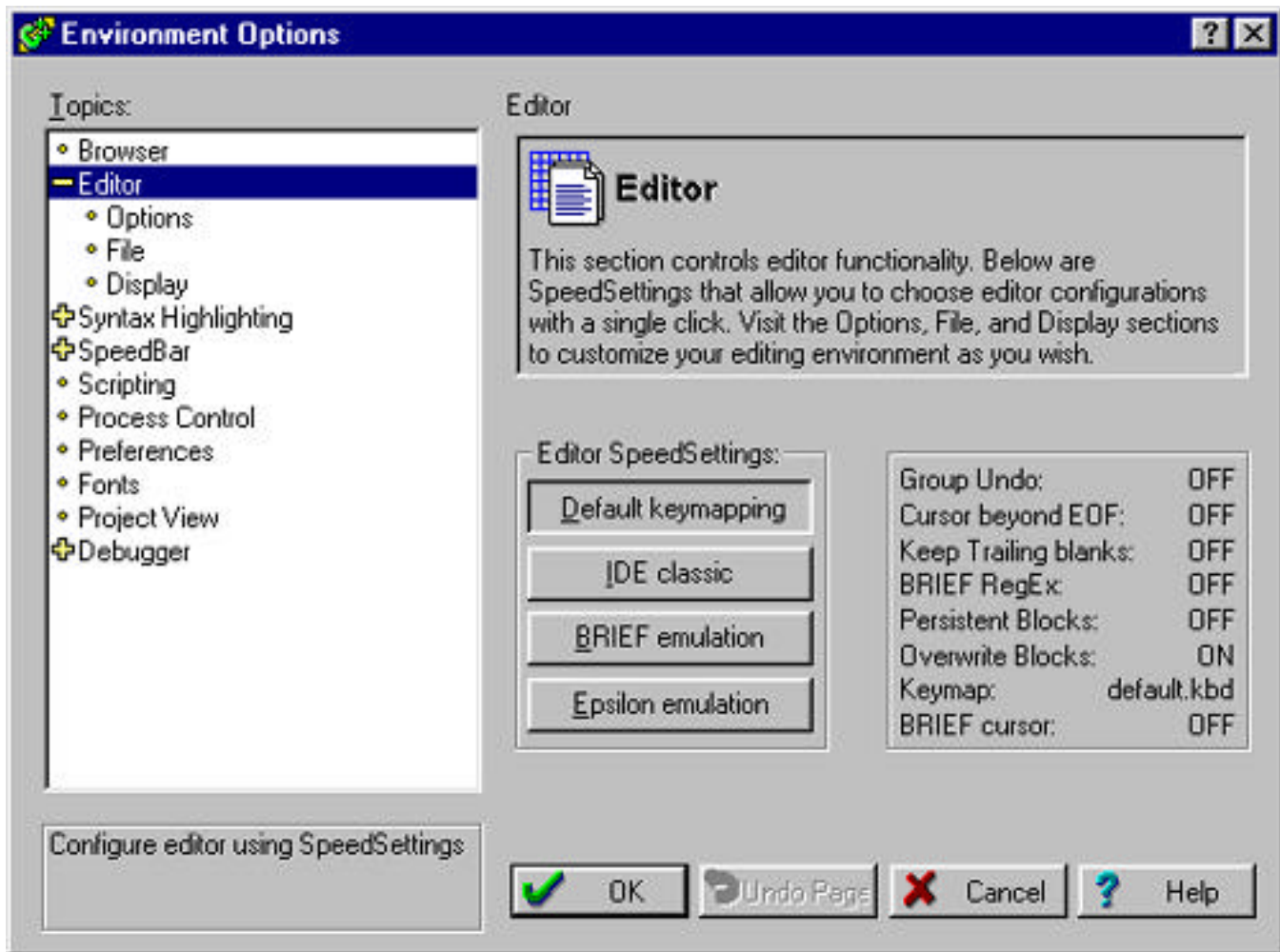
Customizing Paradigm C++

You can configure Paradigm C++ in many ways to create a customized environment that meets your programming needs. For example, you can have Paradigm C++ do tasks automatically (such as saving backups of your files in the Editor windows) or handle special events.

The Environment Options dialog box (accessed with the Options|Environment command) lets you configure the different elements and windows of Paradigm C++. Once you've customized Paradigm C++ to your liking, choose Options|Save, check the options you want to save, then choose *OK*; Paradigm C++ saves your environment settings to a file called PCCONFIG.PCW. By default, the file is saved to the BIN directory in your Paradigm C++ directory tree. This default directory is specified by the DefaultDesktopDir field of your PCW5.INI file, which is located in your Windows directory.

The Environment Options dialog box displays a list of customizable topics on the left and each topic's configurable options on the right. Some topics contain subtopics, indicated by a + next to the topic. For example, the Editor topic has subtopics called Options, File, and Display. To view a topic's subtopics, click the + sign next to the topic; its subtopics appear under it and the + turns to a - (you can then click the - to collapse the list of subtopics). Topics without subtopics appear with a dot next to their name.

Figure 1-8 Environment Options dialog box



This section discusses the following Environment options topics:

- Configuring the Paradigm C++ editor
- Selecting the Syntax highlighting options
- Customizing the SpeedBars
- Setting the Paradigm C++ preferences
- Saving your Paradigm C++ settings



Although this chapter doesn't offer a complete reference to the many selections in the Environment Options dialog box, a complete reference is available by clicking the Help button.

Configuring the Paradigm C++ editor

You can configure the editor so that it looks and behaves like other editors such as Brief and Epsilon. The Paradigm C++ editor uses keyboard mapping files (.KBD files) that set the keyboard shortcuts for the editor and the other windows in Paradigm C++. You can modify this behavior using ObjectScripting. For more information, see "ObjectScripting" the online Help index.

Syntax highlighting

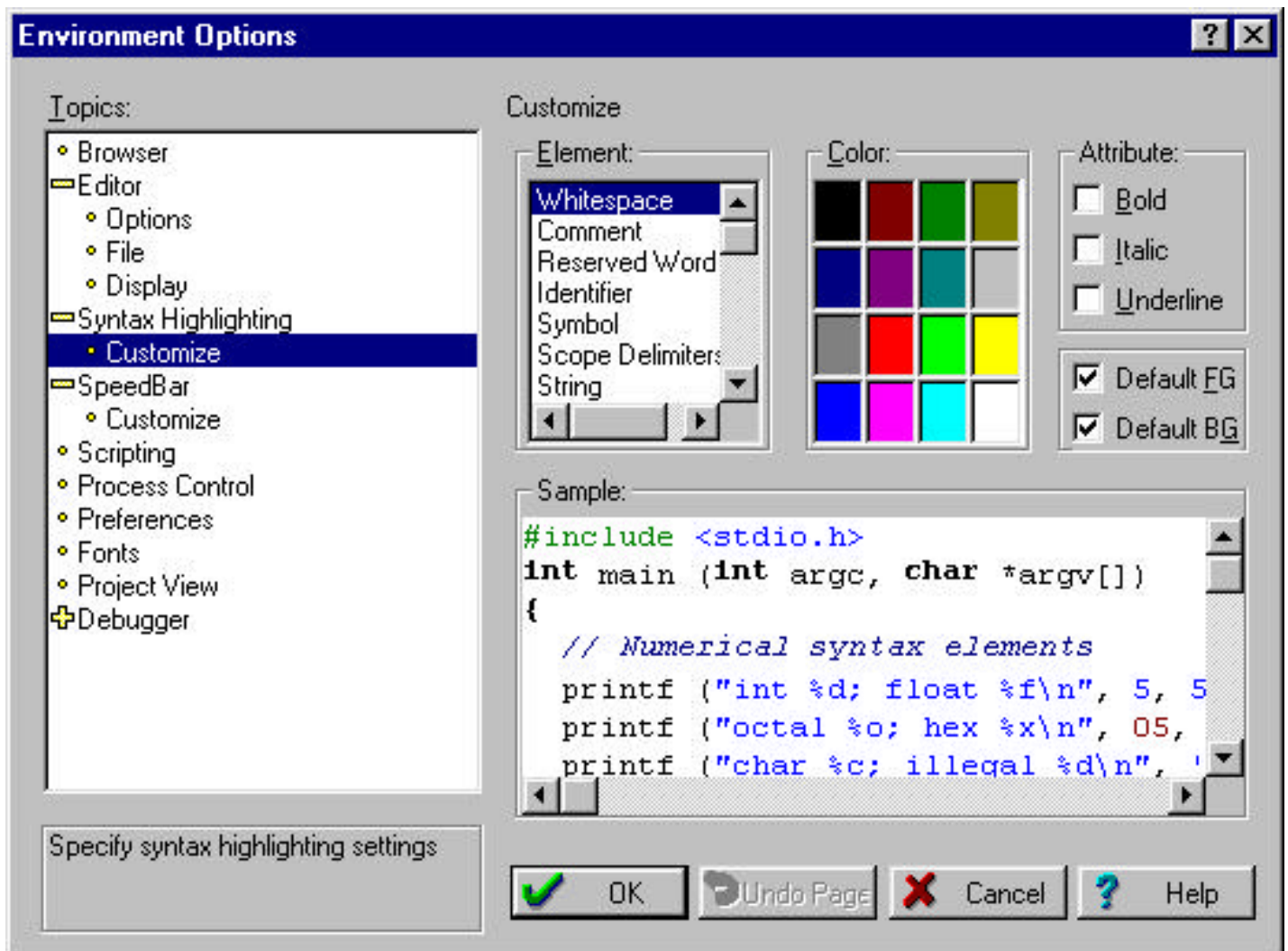
Syntax highlighting lets you define a color and font attribute (such as bold) for certain elements of code. For example, you could display comments in blue and strings in red. Syntax highlighting is on by default.

Syntax highlighting works on files whose extensions are listed in the Syntax Extensions list (by default, these files are .C, .CPP, .H, and .HPP). You can add or delete any extension from this list, but be sure to separate extensions with semicolons.

The Syntax highlighting section displays the default color scheme and four predefined color settings. To use a predefined color scheme,

1. Choose Options|Environment|Syntax highlighting.
2. Choose one of the four predefined color schemes (Defaults, Classic, Twilight, or Ocean) by choosing the Color SpeedSettings; the sample code changes to the color scheme you select.

Figure 1-9 Environment Options Syntax Highlighting dialog



To customize the syntax highlighting colors,

1. Choose Options|Environment, then select the Syntax highlighting topic.

2. Select a predefined color scheme to use as a base for your customized colors.
3. Choose the Customize topic listed under the Syntax highlighting topic. Elements and sample code appear on the right of the Environment Options dialog box.
4. Select an element you want to modify from the list of elements (for example, choose Comment), or click the element in the sample code (this selects the name in the Element list). You might need to scroll the sample code to view more elements.
5. Select a color for the element. The element color in the sample code reflects your selection. Use the left mouse button to select a foreground color for the element (FG appears in the color). Use the right mouse button to select a background color (BG appears in the color). If FB appears in the color, the color is used as both a background and a foreground color.
6. If you want, choose an Attribute (for example, bold).
7. You can check Default FG (foreground) or BG (background) to use the Windows default colors for an element.
8. Repeat steps 2-4 for the elements you want to modify.

To *turn off* syntax highlighting, choose Options|Environment|Syntax highlighting, then uncheck Use Syntax highlighting.

Customizing the SpeedBars

Paradigm C++ uses context-sensitive SpeedBars for all its windows, including Edit, Browser, Debugger, Project Manager, Message, and Desktop windows. When a window has focus, the corresponding SpeedBar appears just below the Menu Bar. Using the Environment Options dialog box, you can customize the SpeedBars for each window so that they include only the buttons you want.

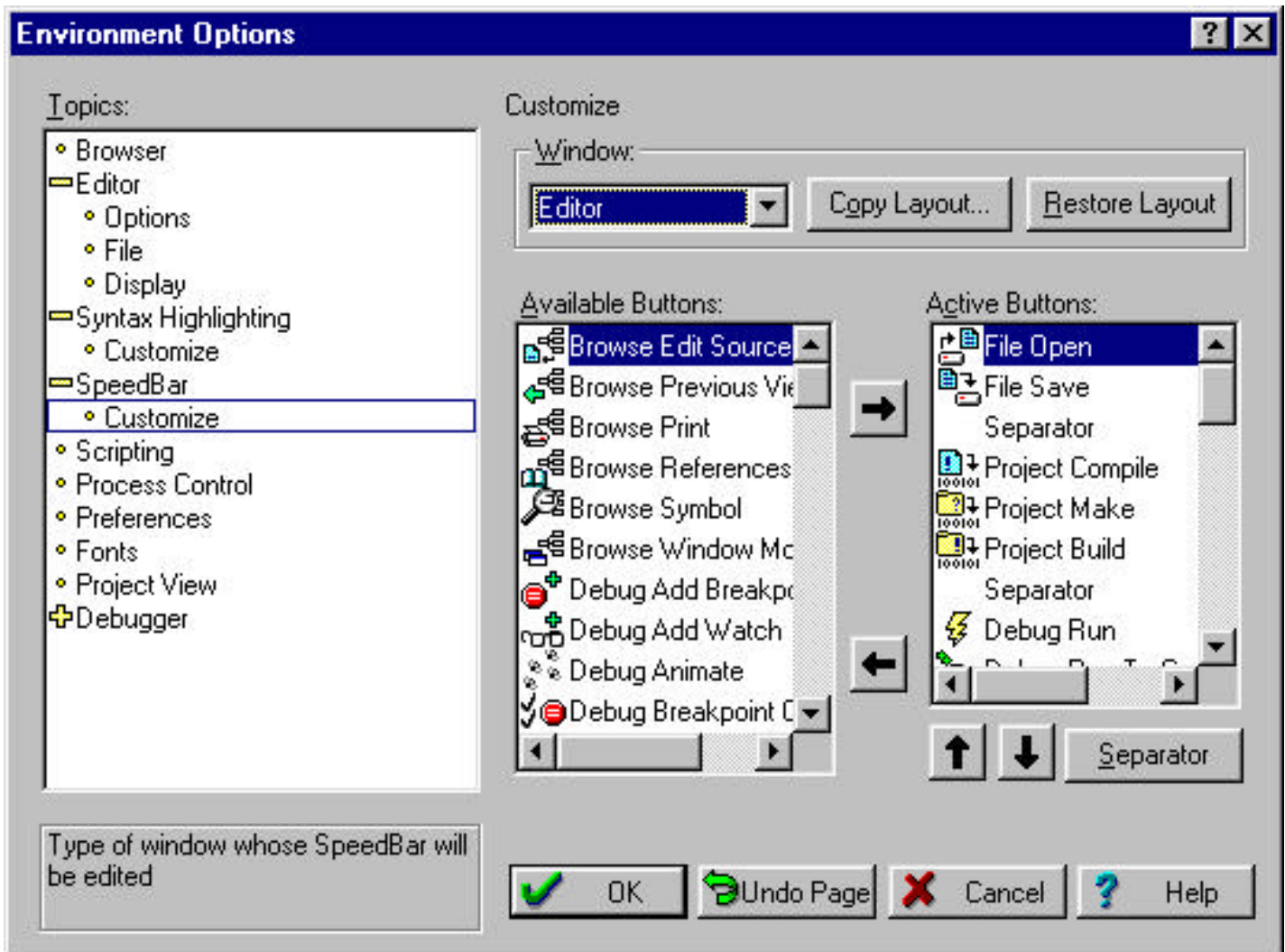
To add or delete buttons from the SpeedBars,

1. Choose Options|Environment from the Paradigm C++ Menu Bar.
2. Choose the SpeedBar topic on the left. The right side of the dialog box displays general options for all SpeedBars.

The options here let you specify if you want to hide or view the SpeedBar, where you want the SpeedBar to appear (on the top or bottom of the Paradigm C++ window), and if you want to use the Flyby Help Hints. If you check Use Flyby Help Hints, Paradigm C++ displays descriptions of the SpeedButtons on the status line when you pass the mouse pointer over a button. If you leave this box unchecked, the hints show on the status line only when you click a SpeedButton.

3. Choose the Customize topic listed under the SpeedBar topic to customize the SpeedBar for a particular window.

Figure 1-10 Environment Options SpeedBar customizing dialog



4. In the Window dialog box, choose the specific window (Edit, Browser, Debugger, Project, Message, or the Paradigm C++ Desktop) whose SpeedBar you want to customize.

The Available Buttons list box displays all the unused buttons that you can add to a particular window's SpeedBar (each button has a name next to it that describes the button's function.). The Active Buttons list displays the buttons that are currently contained in the selected window's SpeedBar.

- *To add a button* to a SpeedBar, double-click the button icon in the Available Buttons list, or select it and click the right-pointing arrow. Paradigm C++ places the button in front of the selected button in the Active Buttons list.
- *To remove a button* from a SpeedBar, double-click the button icon in the Active Buttons list, or select it and click the left-pointing arrow. The button moves to the Available Buttons list.
- *To reorder the button positions* for a SpeedBar, select a button in the Active Buttons list, and use the up and down arrows to move the button within the list. The top button in the list appears on the left side of the SpeedBar and the last button in the list appears on the right side of the SpeedBar.

- To put separator spaces between buttons on the SpeedBar, select a button from the Active Buttons list, and then click the Separator button. The separator is added *before* the selected button.

You can also make all SpeedBars identical by selecting a SpeedBar in the Window list, then pressing the Copy Layout button. A dialog box appears in which you check all the SpeedBars you want to make identical to the selected Speedbar. For example, if you first choose the Editor SpeedBar and then click Copy Layout, the dialog box appears with Editor dimmed. If you then check Project and Message, those SpeedBars will be exactly the same as the Editor SpeedBar.

You can restore any SpeedBar to its original defaults by selecting the SpeedBar in the Windows list, then clicking the Restore Layout button.

Setting Paradigm C++ preferences

The Preferences command lets you customize which of the Paradigm C++ settings you want automatically saved and how you want some Paradigm C++ windows to work.

To set preferences,

1. Choose Options|Environment|Preferences.
2. Check and uncheck the options you want, then choose *OK*. For an explanation of each option, select the option and hit *F1* to access the online Help for that option.

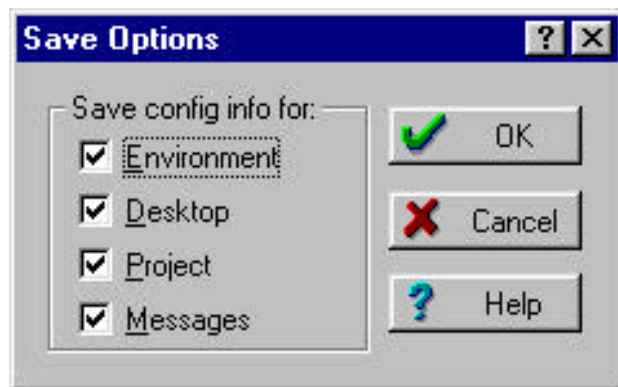
Saving your Paradigm C++ settings

Paradigm C++ automatically saves information when you exit Paradigm C++, use a transfer tool, build or make a project, run the integrated debugger, or close or open a project. You can control which areas of Paradigm C++ get saved from the Preferences topic in the Environment Options dialog box (choose Options|Environment from the main menu).

If you want to save your settings manually, you can do so as follows:

1. Choose Options|Save.

Figure 1-11
Save options
dialog



2. Check **Environment** to save the settings from the Editor, Syntax highlighting, SpeedBar , Browser, and Preferences sections of the Environment Options dialog box. These settings are saved in a file called PCCONFIG.PCW.

3. Check **Desktop** to save information about open windows and their positions. This information is saved to a file called <prjname>.DSW. If you don't have a project open, the information is saved to a file called PCWDEF.DSW.
4. Check **Project** to save the changes to your project (.IDE) file, including build options and node attributes.

Using help in Paradigm C++

Paradigm C++ provides complete online documentation through the Help system. Using Help is a convenient way to get information about language features, compiler options, and any tasks you need to perform while developing applications in Paradigm C++.

Online help organization

The Help system is organized into Help files that include the following documentation:

Table 1-2
Help files

Help file	Description
Using Online Help	Features of Paradigm C++ Help (OPENHELP.HLP)
Paradigm C++ Class Libraries Guide	Programming and reference material (CLASSLIB.HLP)
Paradigm C++ Programmer's Guide	Programming tips and language details (PCPP.HLP)
Paradigm C++ User's Guide	Paradigm C++ tasks, projects, tools (PCW.HLP)
Error Messages and Warnings	Paradigm C++ Error message descriptions (PCERRMSG.HLP)
Tools and Utilities	Command-line tools (PCTOOLS.HLP)
ObjectScripting Guide	Customizing with scripts in Paradigm C++ (SCRIPT.HLP)
Paradigm LOCATE Reference	Reference material for Paradigm LOCATE (LOCATE.HLP)
Paradigm LOCATE Error Messages	Paradigm LOCATE Error messages (LOCERR.HLP)
PDREMOTE/ROM Help	PDREMOTE/ROM Tutorial help (PDREM.HLP)
Paradigm Assembler Help	Assembler options and operators reference (PASM.HLP)
Paradigm C++ SCCS Integration	Source code control system features (SCCS.HLP)
Run-time Library Source Code	Building and customizing tips (RUNTIME.HLP)
PDREMOTE/ROM Source Code	Building and customizing tips (PDREMSRC.HLP)
Paradigm OMFCVT Guide	Features of Paradigm OMFCVT (OMF.HLP)

Some of these files may only be available if you have optional components installed in the Paradigm C++ IDE. Additional files may be available.

Getting help in Paradigm C++

In Paradigm C++, you can get Help in the following ways:

- Context-Sensitive Help (*F1*)
- Contents Screens
- Index
- Keyword Search (*F1* or *Ctrl+F1* in the Edit Window)
- SpeedMenus (in the Help window)
- Contacting Paradigm

Getting context-sensitive help

To access context-sensitive Help for items in Paradigm C++:

1. Select the element you want help on (menu, menu command, an item in a dialog box).
2. Press *F1* or *Ctrl+F1*.

Help buttons are available on many dialog boxes and for most error messages.

Click Help to view information about:

- The entire dialog box
- An error message
- The current group of topics in an Options settings dialog box

Accessing and using contents screens

*To return to a previous topic or Help file, click the **Back** button.*

Each Help Contents offers an entry into a Help system installed with Paradigm C++. From the Contents, select the category of information that best suits your needs, then click on it.

- To display the Master Contents screen, choose **Contents** on the Help menu in Paradigm C++.
- To access the Help Contents from within a topic in the active Help file, click the **Contents** button.
- To access the Help Contents screen of a different Help file installed with Paradigm C++, right-click and select the name of the Help file you want to view.
- To access the Contents of all available Help files, click the Book Shelf button from within the topic of a Help file. Shortcuts to help files are also listed under the Start menu in Programs|Paradigm C++|Help.

You can expand books that appear on the Contents, or jump directly to a topic. To view a topic, click on it.

You can print several topics at once by clicking a book on the Contents and then clicking Print.

Using the index

In Help, click the Index tab to view a list of index entries. Either type the word you're looking for or scroll through the list.

Searching for keywords

Keyword Search gives you direct access to Help about a term in your program. To get help on a term:

1. In the Edit window, place the insertion point on the term you want help on.
2. Use one of the following methods:
 - Press *F1* or *Ctrl+F1*.
 - Choose **Keyword Search** on the Help menu.
 - Choose **Go To Help Topic** on the Edit Window SpeedMenu.
3. One of these events occurs:
 - The topic associated with the term you selected is displayed.

*To return to a previous topic or Help file, click the **Back** button.*

- If more than one topic is available on the term for which you requested Help, the Topics Found dialog box is displayed listing topics associated with the term. Double-click the topic you want to view.
- If no Help is available for the term nearest the insertion point, the index is displayed. You can then select a different searching method to locate a topic associated with that term. The term for which you requested Help appears highlighted in the top box. Click the **Display** button or double-click the term to view the list of topics associated with the term.

Help SpeedMenus

All the Paradigm C++ Help files have SpeedMenus that you access by right-clicking on the mouse. These menus provide quick access to commands for copying or printing a Help topic, or exiting Help.

The SpeedMenu also lists additional Help files containing information related to the current Help file. Right-click and select a Help file from the SpeedMenu. The Contents screen for that Help file is displayed.

Contacting Paradigm

There are several ways to contact Paradigm Systems for technical assistance on Paradigm C++.

Use the Help menu links to access the Paradigm C++ home page, newsgroups, FTP site or to register Paradigm C++.

You can contact Paradigm directly at:

Paradigm Systems
Suite 2214
3301 Country Club Road
Endwell, NY 13760
USA

Sales: 607-748-5966, 800-537-5043
Fax: 607-748-5968
Technical Support: 800-582-0864



Ninety days of free technical support is only available to registered users of Paradigm C++. If you haven't yet done so, take this time to register your products under the Paradigm C++ Help menu or online at <http://www.devtools.com>. Contact Paradigm to purchase a Paradigm SurvivalPak for an additional 12 months of free technical support and quarterly product upgrades.

Managing projects

The Paradigm C++ IDE contains a Project Manager that gives you a visual representation of the files contained in your project. With the Project Manager, you can see exactly what files you're building, the files you're using in the builds, and the options that you've set for the builds.

This chapter covers the following topics, which describe how to use the Project Manager to organize the files in your project:

- Project management
- Using the Project Manager
- Grouping sets of files with Source pools
- Translators, viewers, and tools

What is project management?

As an application grows in size and complexity, it becomes dependent on various intermediate files. Often, source files need to be compiled with different compilers and different sets of compiler options. Even a simple embedded application can have multiple C or C++ source files, with each file type requiring different compilers and different compiler settings.

As your project complexity increases, the need increases for a way to manage the different components in the project. Looking at the files that make up a project, you can see that a project combines one or more *source files* to produce a single *target file*. While target files are usually executable .AXE or .HEX files, source files cover a broader range of file types, including .C, .CPP, and .ASM files. Additionally, many source files have *autodependent files* (files that are automatically included by the source), such as C header files. In larger projects, you are likely to find several targets with scores of sources.

Project management is the organization and management of the source and target files that make up your project. In addition, project management encompasses how and when you employ different tools to translate the source files into your project target files.

Project management tools

Paradigm C++ provides several tools to help you manage your application projects.

Table 2-1
Project
management
tools

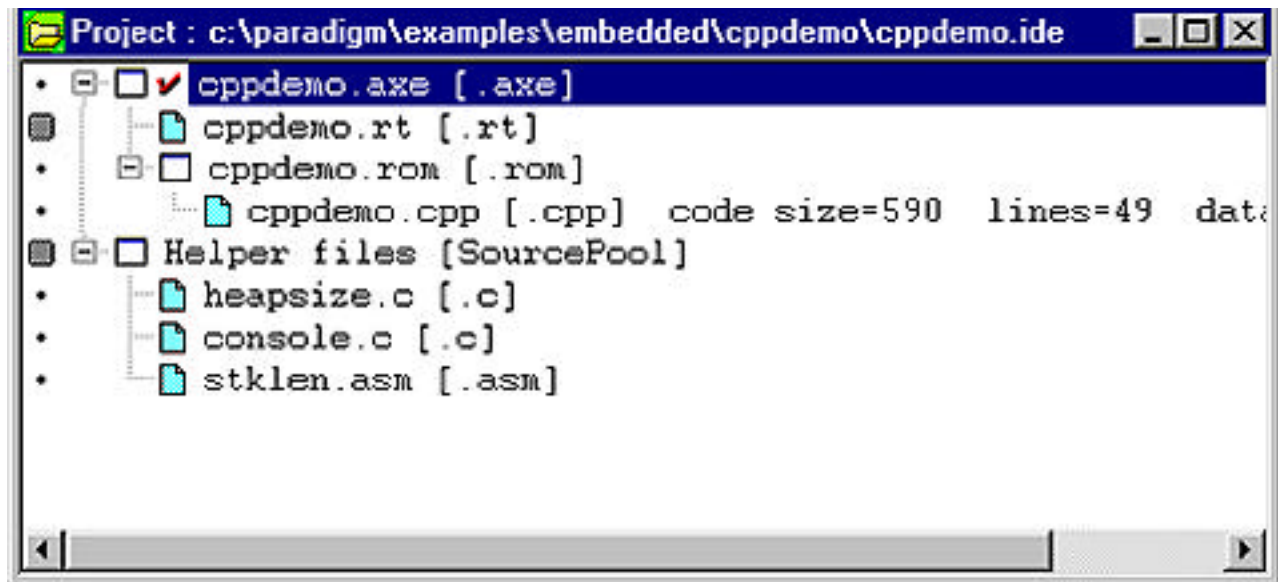
Tools	Description
Project Manager	The Project Manager is the main tool for managing projects in Paradigm C++. Use the View Project command to access the Project Manager, a collapsible/expandable, hierarchical display of the files in your project.
Project menu	The Project menu provides commands to open and close projects, add a new target to a project, and make, build, or compile targets.

Options Hierarchy	The View Options Hierarchy command (located on the Project Tree window SpeedMenu) opens a dialog box that lets you set options for individual project <i>nodes</i> .
Node attributes	The Edit Node Attributes command (located on the Project Tree window SpeedMenu) lets you control how each node is handled by the Project Manager.
Tools	Use the Options Tools command to install, delete, or modify the <i>tools</i> that you use in your projects.
TargetExpert	TargetExpert opens when you create a new project or add a new <i>target node</i> to an existing project. TargetExpert makes available the appropriate platform, model, and library choices based on the type of target you select.

Using the Project Manager

The Project Manager visually organizes all the files in your project in a hierarchy diagram known as the *project tree*. The Project Tree represents each file in your project as a *node* on the tree. The Project Tree is divided into discrete levels where each level contains a single target node. Indented below each target node are the target's *dependencies*-the files used to build the target. To expand and collapse the hierarchy tree, click nodes containing the + and - symbols.

Figure 2-1 Project Tree



The Project Manager uses the following types of nodes to distinguish the different types of files in your project:

The *project node*, located at the top of the Project Tree, represents the entire project. All the files used to build that project appear under the project node (similar to a symbolic target in a makefile). By default, the project node is not displayed in the Project Tree. To display the project node, choose Options|Environment and select Project View from the list of topics, then check Show Project Node.

A *target node* represents a file that is created when its dependent nodes are built. A target can be one of a variety of *target types*, but is usually an .AXE or .LIB file that you are creating from source code. A project can contain many target nodes. For example, in a single project, you might build three separate .AXE files, making three targets in all.

Source nodes refer to the files that are used to build a target. Files such as .C and .CPP are typical source nodes.

A *run-time node* refers to files that the Project Manager uses during the linking stage of your project, such as startup code and .LIB files. The Project Manager adds different run-time nodes depending on the options you specify in TargetExpert. By default, run-time nodes are not displayed by the Project Manager. To view run-time nodes, choose Options|Environment|Project View, then check Show Runtime Nodes.

Autodependency nodes are the files that your program automatically references, such as included header files. By viewing autodependency nodes, you can see the files that source nodes are dependent upon, and you can easily navigate to these files (just double-click the node). By default, the Project Manager does not display Autodependency nodes; you must choose Options|Project|Make, then check Autodependencies: Cache & Display. Note that you must build the project before the Project Manager can display autodependency information).

The Project Manager uses the following color schemes for its nodes:

- Blue nodes represent those that were added by the programmer.
- White nodes indicate project targets.
- Yellow nodes are those that were added programmatically by the compiler (when it posts dependencies and Autodependencies), or by TargetExpert (when it adds nodes based on the target type).

The Project Manager uses special *glyphs* in the left margin to indicate the build attributes of project nodes. To apply build attributes to a node (and for a reference on the different Project Manager glyphs), choose Edit Local Options from the Project Manager SpeedMenu, then select the Build Attributes topic.

In addition to helping you organize your project files, you can use the Project Manager to access source files and build targets.

- To bring a source file into an Edit window, double-click the node in the Project Tree, or highlight the node and either press *Enter* or choose View|Text Edit from the Project Manager SpeedMenu.
- Using the Project Manager to *make* a project is very effective because you can use the Project Manager to *translate* only the files that have changed since the last project build; computer resources are not wasted on unnecessary file updates. (The term "translate" refers to using one file type to create another. For example, the C++ compiler is a translator because it generates .OBJ files from .CPP files.)

There are several ways to customize the build options of the nodes in your project. Maintaining project option and compiling project targets is described in detail in Chapter 3, *Project options*.

Project Manager reference

The Project Tree can be traversed with the mouse or the keyboard.



The Project Manager supports *incremental searching*, so you can quickly find a node by typing the node name. Incremental searching finds the first node in the Project Manager that matches the letters you type. Press *Ctrl+S* to find the next match.

Table 2-2
Project Manager
reference

Task	Keyboard	Mouse
Add Node	Insert	Right Click Add Node

Collapse hierarchy	Minus	Click parent node
Collapse/Expand node	Spacebar	
Copy Node		Ctrl+Left Click Drag
Default action for node	Enter	Double Click
Delete Node	Delete	
Demote a node	Alt+RightArrow	Left Click Drag
End node search	Esc	
Expand hierarchy	+ (Plus)	Click parent node
Expand entire hierarchy	* (asterisk)	
Find a node	Incremental search (start typing)	
Move down in project	DownArrow	Scroll Bar
Move node down	Alt+DownArrow	Left Click Drag
Move node up	Alt+UpArrow	Left Click Drag
Move to bottom of hierarchy	End	Scroll Bar
Move to top of hierarchy	Home	Scroll Bar
Move up in project	UpArrow	Scroll Bar
Open SpeedMenu	Alt+F10	Right Click
Page down	PgDn	Scroll Bar
Page up	PgUp	Scroll Bar
Promote a node	Alt+LeftArrow	
Reference Copy Node		Alt+Left Click Drag
Scroll left	LeftArrow	Scroll Bar
Scroll right	RightArrow	Scroll Bar
Select a node	Up/DownArrow	Left Click
Select Contiguous nodes	Shift UpArrow	Shift Left Click
Select Non-Contiguous nodes		Ctrl Left Click

Creating a project

When you begin to write a new application, the first step is to create a new project to organize your application's files. The command Project|New Project opens the New Target dialog box.

Setting options with the New Target dialog box

When you create a new project, the IDE automatically assigns default file names to the nodes in your project. The following steps show how to change these default settings and how to complete the initial project setup.

1. Type the path and name for the new project into the Project Path And Name input box (the project name must contain eight characters or less). Note that you don't have to type a file extension because the IDE automatically assigns the extension .IDE to all project files.
2. In the Target Name input box, type the name for the first target in your project. This is usually the name of the .AXE or .HEX file that you want to create.



The remaining fields in the New Target dialog box set the options for the first target in the project. These fields are commonly referred to as the *TargetExpert*, since these are the fields contained in the TargetExpert dialog box.

3. Choose the type of target you want to build using the Target Type list. For more information, see "target types" in the online Help index.
4. Choose a platform for your target using the Platform drop-down list. For more information on individual platform types see "target types" in the online Help index.
5. Select the memory model of the target from the Target Model options:
 - **Small** uses different code and data segments, giving you near code and near data.
 - **Medium** gives you near data and far code.
 - **Compact** is the inverse of the Medium model, giving you near code and far data.
 - **Large** gives you far code and far data.
 - **Huge** is the same as Large model, but allows more than 64K of static data.

32-bit targets

- **Win32 Native** - If Protected address mode is chosen under Platform, selecting Win 32 Native will allow you to generate an application to be executed locally on your PC.
 - **Win 32 Embedded** - If Protected address mode is chosen under Platform, selecting Win32 Embedded will allow you to generate an application to be executed on an embedded target.
6. If needed, click the Advanced button to specify the types of source nodes created with your new target (this procedure is described in the following section).
 7. Click OK to accept the settings and close the New Target dialog box. The Project Manager creates the project file, which is denoted with an .IDE extension

When you close the New Target dialog box, the Project Manager draws a graphical representation of your project in the Project window. The Project Manager creates a target node with one or more source nodes below with the project node. After creating the initial target for a project, you can add, delete, or modify the nodes in your project, as described in the following sections.

Specifying the source node types

The Advanced button in the New Target dialog box opens the Advanced Options dialog box. Use this dialog box to set the types of source nodes that the Paradigm C++ IDE creates with a new target node.

Table 2-3
Source node
types

Extension	File Type
.CPP node	Creates a C++ language source node.
.C node	Creates C language source node.
No source node	Creates a Target node that doesn't use a source node. Use this option when you want to create a Source node that uses the same file name as the name of the project. When you create a new target with this option, you must specifically add the source node.
For embedded Win32 applications	
.DEF	Creates a source node that is associated with a Windows module definition file, which is used by the linker.

Opening existing projects

To open an existing project, choose Project|Open Project, then use the file browser to select an existing .IDE or .PRJ project file (.PRJ files are converted to .IDE files when you save the project). If the project opens, but the Project window is not visible, choose View|Project to access the Project window.

Adding nodes

To add a source node to a project:

1. Select any node in the Project Tree under which you want the new node to appear. For example, if you want the new node to appear under the target, select the target node.
2. Press *Ins*, click the button on the SpeedBar, or right-click the node to open the Project window SpeedMenu and then choose Add node.
3. Using the file browser, choose the file or files you want associated with the new node. Alternatively, you can type the name for the file you want to add.
4. Choose OK to confirm your settings.

You can use the Windows File Manager to add one or more source nodes:

1. Open the File Manager and arrange the windows so you can still view the Project window in the Paradigm C++ IDE.
2. In the File Manager, press Ctrl and select the files you want to add as source nodes.
3. Drag the files from the File Manager and drop them on a node in the Project window. The Project Manager automatically adds the source files under the selected node.

Deleting source nodes

Use care when deleting nodes; you cannot undo the deletion.

To delete a node in a project, select the node and press *Del* or choose Delete Node from the SpeedMenu. To delete many nodes, select the ones you want to delete (press *Ctrl* or *Shift* and click the left mouse button to select multiple nodes), then press *Del*. The Project Manager asks if you want to delete the nodes before it proceeds. If you delete an original node, all reference copies of that node are also deleted.

Adding files without relative path information

Because the Project Tree supports drag and drop, you can copy files right from the desktop file manager. Relative path information is included when files are copied. If you move sources or the Paradigm C++ IDE, the relative path information will be incorrect. Here is how to add files to your project without the presence of relative path information:

- Make sure that the Absolute (Options|Project|Make|New Node Path) is turned off (this is the default setting).
- Right-click on the node under which the added files will become children once they are dropped.
- Choose Add Node from the Project Tree SpeedMenu.
- Browse and highlight the file(s), you want to add. (Hold down the *Ctrl* key to select non-contiguous files.)
- After highlight the desired files, shift focus to the input box and capture to the Clipboard (*Ctrl-C*).
- Browse back to the project file location.

- Shift focus to the input box, paste from the Clipboard (*Ctrl-V* or *Shift + Insert*) and choose OK.

Files added to the project by this method do not have relative path information.

Editing source node attributes

Node attributes describe the source node and define the tool that translates it (if applicable). To edit the attributes of a source node:

1. Right-click the source node (or select the node and press *Alt-F10*), then choose Edit Node attributes from the SpeedMenu. The Node attributes dialog box appears.
2. Update the node attributes, then choose OK to confirm your settings.

Node attributes

- **Name** is the file name of the node, without a file extension.
- **Description** is an optional text description of the node.
- **Style Sheet** is the name of the Style sheet the Project Manager uses when it translates that node. If <<None>> is specified, the Project Manager uses the parent options, plus any local overrides set on nodes higher in the Project Tree hierarchy.



If you need to create or edit an existing Style sheet, click the Styles button to access the Style Sheets dialog box.

- **Translator** names the translator used on that node. The Paradigm C++ IDE assigns a default translator for the node type (for example, CppCompile for a .CPP node), which can be overridden using this field.
- **Node type** defines node extension, which in turn defines the available translators for that node.

Adding target nodes to your project

To add a target to a project with the New Target dialog box:

1. Choose Project|New Target, or click the button on the SpeedBar.
2. Type the name for the new target, then choose one of the following target types:
 - **Standard** (default) can be an absolute executable, .LIB, or other file.
 - **Source Pool** is a collection of files that can be referenced in other targets.
3. Choose OK. If the target type is Standard, the TargetExpert dialog box appears so you can further define your target. If the target type is SourcePool, the Target is added to the project and you can add nodes to it immediately.

When you add a new target, it is always appended to the end of the Project Tree.

To view a sample project with two targets, open the file MULTITRG.IDE in the PARADIGM\EXAMPLES\MULTITRG directory. The project contains a text file that describes how to use two or more targets in a project.

With more than one target in a project, you can choose to build a single target, multiple targets, or the whole project.

Use care when deleting target nodes; you cannot undo the deletion.

Deleting target nodes

To delete a target node:

1. Right-click the target node you want to delete (or highlight it and press *Alt-F10*).
2. Choose Delete Node from the SpeedMenu.
3. The Project Manager asks if you're sure you want to delete the target. Choose OK to delete the target and all its dependencies from the project.

You can also delete several nodes by pressing *Ctrl* and clicking the nodes you want to delete, then press *Del*.

Editing target attributes using TargetExpert

Target attributes describe the target. For example, target attributes can describe either a 32-bit Windows DLL or a 16-bit DOS absolute executable. Using TargetExpert, you can modify the attributes for Standard target types. However, you can't change target attributes for SourcePools.

To change a target's attributes:

1. In the Project window, right-click the target node (or select it and press *Alt-F10*), then choose TargetExpert from the SpeedMenu to open the TargetExpert dialog box.



The TargetExpert fields are a subset of the fields in the New Target dialog box.

2. Update the target attributes, then choose OK to confirm your new settings.

Moving nodes within a project

You can move nodes within a project in the following ways:

- By dragging the node to its new location.
- By selecting the node and pressing *Alt* and the arrow keys. This moves the selected node up or down through the *visible nodes*. You can also use *Alt* and the right and left arrow keys to promote and demote nodes through levels of dependencies. For example, if you have a .CPP file dependent that is on a header file (the .H file appears under and right of the .CPP in the project window), you can move the header file to the same level as the .CPP file by selecting the header file and pressing *Alt* → .

Copying nodes in a project

You can copy nodes in your project file either by value or by reference. When you copy nodes by value, the Project Manager makes an identical, but *separate*, copy of the node in the location you specify. The nodes you copy inherit all the attributes from the original node, and you have the ability to modify any of the copied node's attributes.

When you copy nodes by reference, you simply point to one node from a different location in the project; a reference copy is not distinct from the original node. If you modify the structure of the original node, the reference copy is also modified. However, a reference copy does not inherit the options of the original node; you're free to attach Style Sheets and override options in the copied node without affecting the original node.

To copy project nodes,

1. Select a group of nodes you want to copy (press *Shift* or *Ctrl* and click to select modify nodes). You don't need to select the node's dependents because they are copied automatically.

2. Hold down the *Ctrl* key and drag the selected nodes to the new location to copy *by value*.

Or

Press the *Alt* key and drag the selected nodes to the new location to copy *by reference*.

When you release the mouse button, the copied node appears. If you reference-copied the node, it will appear in a *lighter* font. At this point, if you've copied by value, you can edit either the original or the copied nodes without changing other nodes in the project. If you reference-copied, and you edit the original node (such as adding or deleting dependents), all referenced copies are updated.



You cannot add to, delete, or modify nodes that have been copied by reference; to modify nodes copied by reference, you must edit the master copy. If you delete an original node, *all* reference copies to that node are also deleted. You cannot undo this deletion.

Converting project files into makefiles

Using the Paradigm C++ IDE, you can convert project files (.IDE files) into makefiles (.MAK files). To convert a project file to a makefile:

1. Open the project file you want to convert.
2. Choose Project|Generate Makefile. The Paradigm C++ IDE generates a makefile with the same name as the project file, but with the extension .MAK, and places it in the edit buffer. The Paradigm C++ IDE displays the new makefile in an Edit window.
3. Choose File|Save to save your new makefile.

Customizing the Project window

By default, the Project window displays target nodes and source nodes. To control the display of nodes and options:

1. Choose Options|Environment to open the Environment Options dialog box, then choose Project View. The right side of the dialog box displays the Project View options.
2. Check or uncheck the options you want. A sample node called WHELLO changes as you select or deselect options. This sample shows you how all nodes appear in the Project window.
 - **Build translator** displays the translator used on the node.
 - **Code size** displays the total size of code segments. This information appears only after the node has been compiled.
 - **Data size** displays the size of the data segment in bytes. This information appears only after the node has been compiled.
 - **Description** displays the optional description of the node in the Project Tree. Type the description using the Edit node attributes dialog box from the Project Manager SpeedMenu.
 - **Location** lists the path to the source file associated with the node.
 - **Connection** displays the name of the target connection used for the node. This only applies to target nodes that support a debugger connection.
 - **Number of lines** displays the number of lines of code in the file associated with the node. This information appears only after you compile the code.

- **Node type** describes the type of node (for example, .cpp, or .c).
 - **Style Sheet** names the Style Sheet attached with a node.
 - **Output** names the path and file name that is created when the node is translated. For example, a .CPP node creates an .OBJ file.
 - **Show run-time nodes** displays the nodes the Project Manager uses when the project is built. For example, it lists startup code and libraries.
 - **Show Project Node** displays the project node, of which all targets are dependents.
3. Click OK to close the Environment Options dialog box.
 4. To save your project customizations, choose Options|Save, then check Project. Note that you can save different option sets with the different projects you work on.

Grouping sets of files with Source Pools

What is a Source Pool? A *Source Pool* is a collection of nodes that can be referenced by multiple target nodes. When a Target node references a Source Pool, the nodes in the Source Pool take on the options and target attributes of the target. Because Source Pools let you create different targets using a common set of source nodes, it is easy to maintain the files that the targets use. For example, with Source Pools, you can create both 16- and 32-bit applications using a single set of source nodes. Then, when you add or delete from the Source Pool, you don't have worry about updating all your target nodes; they're updated automatically through the reference to the Source Pool.

You can also use Source Pools when you have several header files that you need to include throughout your project. If you place the header files in a Source Pool, you can reference them wherever you need them in your project. Then, you only have to update the original Source Pool when you need to make changes to the group of header files; if you add a new header file to the Source Pool, all the referenced copies are automatically updated.

Source Pools are also useful when you want to assign a single Style Sheet to multiple nodes. For example, if three targets in a project need to use the same Style Sheet, you can reference a Source Pool that contains the Style Sheet instead of attaching the same Style Sheet to each individual node. Then, if you need to update the Style Sheet (for example, if you want to change from compiling with debug information to compiling without it), you can update all the targets by modifying the single Style Sheet. You can also use Source Pools to apply custom tools to project nodes. For more information, see "Source Pools" in the online Help index.

Creating a Source Pool

When you create a Source Pool, you create a target node with a group of nodes under it. However, the target node of the Source Pool cannot be compiled—to compile the nodes in a Source Pool, you must copy the Source Pool to a another target node. Source Pools work to your best advantage when you copy them by *reference*.

To create a Source Pool

1. In your project, create a new target node by choose Project|New Target.
2. Type the name for the Source Pool in the Target Name.
3. Select Source Pool from the Type list and press OK to create a Source Pool target node in your project.

4. Select the new Source Pool in the Project Tree, then press *Ins* to open the Add To Project List dialog box.
5. Select the source files you want, then press OK to add them to the Source Pool.
6. Copy the Source Pool by reference by holding down the *Alt* key and dragging the Source Pool to the target nodes you want.



To see a working example of Source Pools, open the sample project called SRCPOOL.IDE in the PARADIGM\EXAMPLES\SRCPOOL directory. The project file includes a text file that describes how the Source Pool is used in the example.

Translators, viewers, and tools

Translators, viewers, and tools are internal and external programs that are available to you through the Paradigm C++ IDE.

- **Translators** are programs that create one file type from another. For example, the C++ compiler is a translator that creates .OBJ files from .CPP files; the linker is a translator that creates .AXE files from .OBJ, .LIB, and .DEF files.
- **Viewers** are programs that let you examine the contents of a selected node. For example, an editor is a viewer that lets you examine the source code of a .CPP file.
- **Tools** are programs that help you create and test your applications. The external AXE utility is an example of a programming tool.

The Paradigm C++ IDE associates each node in a project with different translators or viewers, depending on the file extension of the node. Although each node can be associated with several different translators or viewers, each node is associated with a *single* default translator or viewer. This is how the Paradigm C++ IDE knows to open the Edit window when you double-click a .CPP node (double-clicking a node invokes the default viewer on the node).

To see the default node type (determined by file extension) for a specific translator or viewer:

1. Choose Options|Tools to open the Tools dialog box.
2. Select the item you want to inspect from the Tools list.
3. Choose Edit to access the Tools Options dialog box.
4. Choose Advanced to access the Tool Advanced Options dialog box, then inspect the Default For text box.

When you right-click a node, you'll find that some source nodes have a Special command on the SpeedMenu. This command lists the alternative translators that are available for the node type selected. For example, the commands C To Assembler, C++ To Assembler, and Preprocess appear on the Special menu of a .CPP node. The command Implib appears if you selected a .DLL node. Using the Special command, you can invoke any translator that is available for a selected node type. Also, by selecting a source node in the Project Tree and choosing Edit Node Attributes from the SpeedMenu, you can reassign the default translator for the selected node.

Adding translators and viewers

The Tools dialog box displays the default set of translators, tools, and viewers. The following steps show how to add an item to this list of programs:

1. Choose Options|Tools to access the Tool Options dialog box. This dialog box displays the default list of translators, tools, and viewers.
2. Choose New to add a new program to the Tools list (to modify a program that is already listed, select the tool, then choose Edit).
3. Set the following option in the Tools Options dialog box:
 - **Name** is a description of the item you're adding. This is the placed on the Tool list.
 - **Path** is the path and executable program name. You can use the Browse button to complete this selection.
 - **Command-line** holds any command-line options, *transfer macros*, and the Paradigm C++ IDE filters you want to pass to the program. For more information, see "transfer macros" in the online Help index. (Try using **\$PROMPT** if you want to experiment with transfer macros.) the Paradigm C++ IDE filters are .DLL files that let tools interface with the Paradigm C++ IDE (for example, the GrepFile tool uses a filter to output text to the Message window). To see transfer macros and filters in use, choose Options|Tools, then select GrepFiles and choose Edit.
 - **Menu Text** appears on SpeedMenus and on the Tools menu. If you want to assign a shortcut key to your menu text, precede the shortcut letter with an ampersand (&) - this letter appears underlined in the menu. For example, &File assigns the letter F as the shortcut key for File. If you want an ampersand to appear in your menu text, use two ampersands (&&Up&date appears as &Up&date in the menu).



You *must* supply Menu Text if you want the program item to appear on the SpeedMenu or Tools menu.

- **Help Hint** is descriptive text that appears in the status line of the Tools dialog box when you select the program item.
4. Open the Advanced Options dialog box (choose Advanced) to set the options for your new program. Depending on the Tool type you choose (Simple Transfer, Translator, or Viewer), different fields become available. If you create a Translator, the program becomes available for make and build processes.
 - **Place On Tool Menu** adds the item to the Tools menu.
 - **Place On SpeedMenu** adds a viewer or translator to the associated SpeedMenu.
 - **Target Translator** available for translators and viewers. For translators, this field specifies whether the program produces a final target (such as an .AXE file) or an intermediate file (such as an .OBJ or .I file). If you check this box, the translator produced a final target that is saved to the directory you specify in the Final text box (choose Project|Options|Directories). If you don't check Target Translator, the translated file is saved in the directory you specify in the Intermediate text box.

For viewers, Target Translator specifies that the viewer works only on nodes that have been translated (such as .OBJ or .AXE files); the node has to be translated before you can view it.

- **Translate From** defines the node types (determined by file extension) that a translator can translate. To specify multiple node types, use a semicolon to separate file extensions.

When you enter a file extension in this field, the Project Manager adds the translator to the Special menu of the project nodes that have that file extension. When you

choose Special from the Project Manager SpeedMenu, the Project Manager displays all the available translators for that node type. However, it is important that each node type can have only a single, default translator (see the description for Default For).

To see how this works, look at the tool CppCompile (choose Options|Tools, double-click CppCompile, then click Advanced). The Tool Advanced Options dialog box shows that the C++ compiler is a translator for .CPP, .C, .CAS, and .H files. If you have a source node with a .C extension, CppCompile appears on the Special menu when you right-click the node and choose Special.

- **Translate To** defines the extension of the file that the translator generates.
- **Applies To** is similar to Translate From field, except that it's used for viewers instead of translators.
- **Default For** changes the Paradigm C++ IDE's default translator or viewer for the file types you specify. Type the file extensions (separating each with a semicolon) for the file types whose default you want to override.

5. Choose OK twice to confirm your settings, then close the Tools dialog box.

Your new tool has now been added to the Tools list of the associated project, and to the Tools menu or SpeedMenu, depending on where you chose to add the item. If you added the item to the Tool menu, you can check the addition by choosing Tools from the main menu; the new program name appears on the Tools list.

Although the Project Manager lets you define your own Tools items, these items apply only to the project that you add them to; they aren't added as permanent parts of the Paradigm C++ IDE. However, translators, viewers, and tools can be *passed* to new and existing projects by sharing the Style Sheets of the projects.

Project options

After you create a project file and write the code for the source nodes in your project, you need to set the options for the different project nodes before you can compile the project. This chapter describes how to set options in a project, how to view the options you set, how to compile a project, and how to use the Message window to view and fix compile-time errors. In addition, this chapter contains a complete reference to the compiler and linker options that can be set from the Paradigm C++ IDE.

Setting project options

This section explains how to set, view, and manage project options.

Project options tell the Paradigm C++ IDE how to compile and link the nodes in your project to form the targets you need. The settings of the project options can indicate whether or not to generate debugging information, where to look for source code, what types of compiler optimizations you want to use, and so on.

The Project Manager lets you set project options in two different ways:

- You can attach Style Sheets to your project nodes.
- You can override the settings in a Style Sheet using local overrides.

Style Sheets group a collection of option settings into a single unit. Once a Style Sheet is created, you can attach it to a node, a group of nodes, or an entire project. Local overrides are settings that take precedence over Style Sheet settings at the node level.

Using Style Sheets

A Style Sheet is a group of option settings. In your project, for example, you might want to compile .C files with one set of options and .CPP files with another, or you might want to build one target with debugging information, and another one without it. Style Sheets make it easy to view and maintain the settings of your project options. Option settings control how *target nodes* in your project are built. You can attach Style Sheets to entire projects or to individual nodes in a project. You can attach one or more Style Sheets to your entire project or assign one or more Style Sheets to individual nodes in your project.

To view the options that can be incorporated into a Style Sheet, open the Project Options dialog box by choosing Options|Project. This dialog box contains a hierarchical list of topics on the left, with the options that relate to each topic listed on the right. To expand and collapse the Topic list, click the + and - icons to the left of the topic listings.

To see an example of how Style Sheets are used, open the STYLESHT.IDE project file located in the PARADIGM\EXAMPLES directory. This file uses a different Style Sheet for each of its two versions of the application and also contains a text file that explains the use of Style Sheets.

Predefines Style Sheets

The Project Manager contains several predefined Style Sheets that you can attach to any node in your project. You can also customize a predefined Style Sheet to meet the special needs of your projects.

To inspect the predefined Style Sheets, choose Options|Style Sheets on the main menu (or click the Styles button on the Edit Node Attributes dialog box). This opens the Style Sheets dialog box where you can create, compose, copy, edit, rename, or delete from the list of Style Sheets that are available for your project. Predefined Style Sheets are listed on the left with the description of the selected Style Sheet on the right.

The default project options

When you initially create a project, it inherits the Style Sheet known as the Default Project Options. If some components in your project require different settings, you can attach different Style Sheets to those nodes. If different nodes in your project require different option settings, you should override the default option settings by attaching different Style Sheets to the nodes in your project.



Be careful when you use the Options/Project command to modify option settings; if your project contains more than a single target node, the changes you make always modify the project's Default Project Options (regardless of the node you have selected when you choose the command). Because of this, all targets in your project inherit the changes you make when you use the Options/Project command. In addition, if you modify project options when you don't have a project loaded, your modifications update the Default Project Options Style Sheet; the projects you later create will inherit these new default settings. If you need to revert to the Paradigm C++ IDE's factory default settings, delete the file PCWDEF.PCW (located in the Paradigm C++ IDE BIN directory), then open and close the Paradigm C++ IDE to create a new file.

Managing Style Sheets

The buttons at the bottom of the Style Sheets dialog box let you create, compose, copy, edit, rename, and delete user-defined Style Sheets.

Create Create lets you design a new Style Sheet for the currently loaded project. To create a Style Sheet:

Choose the Create button, then enter a name for your new Style Sheet into the Create Style Sheet dialog box. Choose OK to add the new Style Sheet to the Available Style Sheets list.

Compose Compose lets you create a Style Sheet that contains the combined options from one or more Style Sheets. To compose a Style Sheet:

1. Create a new Style Sheet using the Create button.
2. Select the new Style Sheet in the Available Style Sheets list, then click Compose.
3. Select the Style Sheet you want included in your new Style Sheet from the Available Style Sheets list, then move the Style Sheet to the Composite Style Sheets list by double-clicking it or by clicking the → button. (You can also remove Style Sheets from the Composite Style Sheet list by selecting a Style Sheet there and clicking ←.)
4. Continue modifying the composed Style Sheet, then choose OK when you're finished.



You cannot edit the option settings in a composed Style Sheet. However, you can edit the option settings in the Style Sheets contained in the composed Style Sheet, which affects the settings in the composed Style Sheet.

- Copy** Copy lets you create a new Style Sheet from an existing one. When you choose Copy, you're prompted for the new Style Sheet's name. Enter the new name, then choose OK to make an exact copy of the selected Style Sheet. Copying is a fast way to create a Style Sheet that closely resembles another-you only have to change the options you want.
- Edit to change any of the copied options. Copying is a fast way to create a Style Sheet that closely resembles another-you only have to change the options you want.
- Edit** Edit lets you modify the option settings of an existing Style Sheet, including any predefined Style Sheet.
- Rename** Rename lets you rename a selected Style Sheet.
- Delete** Delete lets you remove an unwanted Style Sheet. (This action cannot be reversed.)

Attaching Style Sheets to nodes

Sometimes different nodes in a project need to be built with option settings that are different than those in the project Style Sheet. For example, you might want to compile .C files with one set of options but .CPP files with another. Or, you might want to build one target with 16-bit options and another with 32-bit options.

To attach an existing Style Sheet to a project node:

1. Right-click the node in the Project Tree (or select it and press *Alt-F10*).
2. Choose Edit node Attributes from the SpeedMenu. The Node Attributes dialog box appears.
3. Select a Style Sheet from the drop-down box, then choose OK.

When you attach a Style Sheet to a node, all child nodes of that node inherit the settings of the selected Style Sheet. To change the settings of a child node, attach a different Style Sheet, or override an option setting using a local override.



Although you can attach only a single Style Sheet to a project node, one Style Sheet can be composed of several different Style Sheets.

Sharing style sheets between projects

There are two ways to share Style Sheet between projects:

- inheriting style sheets from another project
- editing the .PDL file associated with a project

When you create a custom Style Sheet, that Style Sheet remains with the project for which it was created; it doesn't get added to the list of predefined Style Sheets. However, if you want a new project to use one of your custom Style Sheets or user-defined tools, you can do so by letting a new project inherit settings from another project.

Before a project can inherit the settings of another project, you must modify the PCW5.INI file that resides in your Windows directory. If the file doesn't contain an inherit setting, then you must add the settings to the file as follows:

```
[Project]
;To have new projects inherit settings from the Default Project
Settings (default) ;
inherit=0

;To have new projects inherit settings from currently open project:
inherit=1
```

```
;To have new projects inherit factory default settings:  
inherit=2
```

To pass Style Sheets or user-defined tools from one project to a new project:

1. Modify PCW5.INI so that `inherit=1`.
2. Open the project that contains the Style Sheet or tools you want to share.
3. Choose Project|New Project.

When the new project is created, it inherits the Style Sheets and user-defined tools of the project that was open when you chose Project|New Project.

Project Description Language files

You can also share Style Sheets across projects by editing the Project Description Language files (.PDL) associated with your projects. When you save a project, you can instruct the Paradigm C++ IDE to create a .PDL file that has the same file name as the project's Paradigm C++ IDE file. Likewise, when you open a project you can instruct the Paradigm C++ IDE to read the project's .PDL file. Because a .PDL file contains information about the Style Sheets and tools used in a project, you can edit a project's .PDL file so that it uses the Style Sheet and tools of your choosing.



Be careful if you choose to edit .PDL files. If a .PDL file is corrupted, the Project Manager will not be able to read it. You may want to make a backup copy of the .PDL file before you begin making changes.

If you plan to use .PDL files to share Style Sheets and tools, you must first ensure that the Paradigm C++ IDE creates and reads the files. To do so, open the PCW5.INI file (found in your Windows directory) and add the following settings to the [Project] section of the .INI file:

```
[Project]  
saveastext=1  
readastext=1
```



The saveastext setting tells the Paradigm C++ IDE to save a .PDL file whenever a project is saved. The readastext setting tells the Paradigm C++ IDE to update an .IDE file if its associated .PDL file is newer than the .IDE file.

To share Style Sheets or user-defined tools between projects:

1. Modify your PCW5.INI file as just described.
2. Open the project that you need to transfer Style Sheets or tools to, then close the project (choose Project|Close Project). This creates a .PDL file for the project.
3. Open the project that contains the Style Sheets or tools you want to share (the .PDL file name matches the file name of the .IDE file).
4. Using the text editor, open the .PDL file containing the Style Sheet or tools you want to share (the .PDL file name matches the file name of the .IDE file).
5. Search for Style Sheet's name. For example, if you created a Style Sheet called MYSTYLE, you'll see a section in the .PDL file that starts { `StyleSheet = "MYSTYLE"`.
6. Copy all the text from the beginning brace to the ending brace. If needed, you can copy more than one Style Sheet.



To share a user-defined tool, copy the section that reads `Subsystem=<tool>`.

7. Open the .PDL file that is to receive the Style Sheet.
8. Find the section for Style Sheets, then paste the copied text to the end of the existing Style Sheet list.
9. Save the .PDL file that received the copied Style Sheet.
10. Open the project that received the copied Style Sheet to update the project's Style Sheets and tools from the .PDL file.

After transferring Style Sheets, it is a good idea to reset the `saveastext` and the `readastext` settings in the PCW5.INI file to 0. This tells the Paradigm C++ IDE to not save to or update from .PDL files.

Setting local overrides

Inherited options or Style Sheet options can be overridden at the node level using local overrides. Local overrides are useful when a node's option settings must differ from its associated Style Sheet by one or two settings. Set options for an individual node by selecting Edit Local Options on the Project Tree window SpeedMenu or by selecting Edit Local Options from the Edit Local Options on the Edit window SpeedMenu when no project is loaded. The local options dialog box displays where the node is located in the Project Manager and allows you to set options for that node.

Once options have been set, they become local overrides associated with the node. Local Override are useful when you use a Style Sheet (perhaps inherited from a parent node).



Although the local overrides make it easy to set options for individual nodes, they have the disadvantage of being difficult to track. While the Options Hierarchy dialog box displays the Style Sheet and local override settings for a selected node, you must examine each individual node to see which ones have been overridden. Because of this, it's recommended that you use separate Style Sheets for nodes that require different option settings, and use local overrides only in special cases.

To override an option setting:

1. Choose the node whose settings you want to override.
2. Right-click the node (or press *Alt-F10*) and choose Edit Local Options from the SpeedMenu. The Options dialog box (which is similar to the Project Options dialog box) appears and displays the settings for that node.
3. Select the option you want to override. The Paradigm C++ IDE automatically checks the Local Override box whenever you modify a Style Sheet setting.
4. Choose OK to confirm you new settings.



The Local Override check box is enabled only when an option within a topic is selected otherwise, the check box is grayed. When you select an option (using Tab, or by clicking and dragging the mouse off the option), the Local Override check box shows the status of the selected option. Because of this, you must individually select each option in a topic to see which ones have been overridden locally. If you choose an option (by clicking it, or by selecting it and pressing *Enter*), you change its setting, which always causes the Local override check box to be checked.

To undo an override:

1. Right-click the node whose setting you want to modify, then choose Edit local options from the SpeedMenu.
2. In the Options dialog box, select the topic that contains the overridden setting.

When you select a topic page that has a locally overridden option, the Project Manager enables the Undo Page button.

3. Select the option (using Tab, or by clicking and dragging the mouse off the option) whose local override you want to undo; the Local Override checkbox will be checked.
4. Click the Local Override check box to undo the override; the option will revert to the default Style Sheet setting. To revert the entire topic to the settings contained in the associated Style Sheet, choose the Undo Page button.
5. Choose OK to confirm your modifications.

View project options

Because each node can have its own Style Sheet *and* you can override the option in the Style Sheet, you need a quick way to view the option settings for each node.

To view option settings for the nodes in you project:

1. Right-click any node in the Project window and choose View Options Hierarchy, or click the button on the SpeedBar.

The Options Hierarchy dialog box appears, listing the nodes in the project on the left and the options that each node uses on the right. You can expand and collapse the list of nodes in the dialog box just like you can in the Project window, however, Autodependency nodes do not appear.

An option that's surrounded by double-asterisks (**) in the Options listing indicates that the option is overridden (by either a Style Sheet or local override) by a dependent node located farther down in the Options listing. (The asterisks display only when you select the node where the option is overridden.)

2. When you select a node in the Project Options At list, its setting appears to the right in the Options list.

The Options list displays components of the project in square brackets. At the top of the list, you'll see the name of the project followed by its Default Project Options. Below this is the name of the target associated with the node you've selected. If the node has a Style Sheet associated with it, it is displayed beneath the node (also in brackets), along with the settings of the Style Sheet. If you've overridden any settings, these are displayed beneath the [Node overrides] listing. The Options list displays the setting for all the ancestors of the node selected in the Project Tree.

3. If you want to edit an option, double-click the option in the Option list, or select it and click Edit. Whenever you edit options in this manner, the modifications become local overrides.
4. When you finish viewing your project's option settings, choose *Close*.

Compiling projects

There are two basic ways to compile projects: built and make. Build compiles and links all the nodes in a project, regardless of file dates. Make compares the time stamp of the target with the time stamps of all the files used to build target. Make then compiles and links only those nodes necessary to bring the target up to date.

To compile a project, open the project using the Project|Open command, then choose either Compile, Make All, or Build All from the Project menu (note that the SpeedBar has three similar looking buttons that correspond to these Project Menu commands).

- Compile (*Alt-F9*) builds the code in the currently active Edit window. If a Project window is selected, all the selected nodes in the project are translated; child nodes aren't translated unless they're selected.
- Make all (*F9*) translates all the out-of-date nodes in a project. If a project is not open, the file contained in the active Edit window buffer is built.

When you choose Make All, the Project Manager moves down the Project Tree until it finds a node with no dependents. The Project Manager then compares the node's date and time against the date and time of the node's parent. The Project Manager translates the node only if the child node is newer than the parent node. The Project Manager then moves up the Project Tree and checks the next node's date and time. In this way, the Project Manager recurses through the Project Tree, translating only those nodes that have been updated since the last compile.

- Build All translates all nodes in a project - even if they are up-to-date. Build All always starts at the project node and builds each successive target down the project. Choose *Cancel* to stop a build.

When you choose Build All, the Project Manager starts at the first target and works down the Project Tree until it comes to a node with no dependents. The Project Manager compiles that node first (and other nodes on the same level), then works back up the Project Tree, compiling and linking all nodes needed to create the target. This process is then repeated down the Project Tree, until all the targets have been updated.

For example, if you have a project with an .AXE target that is dependent on two separate .OBJ files, the Project Manager creates the first .OBJ file by compiling all its dependents. It then creates the next .OBJ file. Once a target node's dependents are created, it can compile or link the target node. In this case, the Project Manager will link the two .OBJ files (and any run-time nodes) to create the final .AXE.

Compiling part of a project

You can compile part of a project several ways:

- Translate an individual node.
- Build a node and its dependents.
- Make a node and its dependents.
- Select several nodes and compile.

To translate an individual node:

1. Select the node you want to translate.
2. Choose Project|Compile from the main menu or choose the default translation command from the SpeedMenu. For example, if you've selected a .CPP file, the node SpeedMenu contains a C++ Compile command, which compiles only the selected node.

To build a node and its dependents:

1. Choose the node you want to build.
2. Right-click the node (or press *Alt-F10*) and choose Build Node from the SpeedMenu. All the dependent nodes are built regardless of whether they're out-of-date.

To make a node and its dependents:

1. Choose the node you want to build.

2. Right-click the node (or press *Alt-F10*) and choose Make node from the SpeedMenu. This command compiles only the dependent nodes whose source files are newer than their associated target files.

To compile several selected nodes:

1. Select the project nodes you want to compile by pressing *Ctrl* and clicking the desired project nodes. (The nodes must be the same file type, such as .CPP).
2. Choose Make Node or Build Node from the Project Manager SpeedMenu to compile the selected nodes.

Fixing compile-time errors

Compile-time errors, or *syntax* errors, occur when your code violates a syntax rule of the language you're programming in; the C++ compiler cannot compile your program unless it contains valid language statements. If your compiler encounters a syntax error while compiling your code, the Message window opens and displays the type of error or warning it encountered. By choosing Options|Environment|Preferences, you can specify if old messages should be preserved or deleted between calls to different programming tools (such as compiler, or GREP). Check Save Old Messages if you want the Message window to retain its current listing of messages when you run a tool.

To clear the Message window, choose Remove All Messages from the Message window SpeedMenu.

Viewing errors

To view the code that caused a compiler error or warning, select the message in the Message window; the Paradigm C++ IDE updates the Edit window so that it displays the location in your code where the error or warning occurred (this is called Automatic Error Tracking). If the file containing the error isn't loaded in an Edit window, press Spacebar to load the file (you can also load the file by pressing *Alt-F10*, then choosing View Source from the SpeedMenu). When you view errors in this manner, the Message window remains selected so you can navigate from message to message. To open or view the Message window, click the button on the SpeedMenu, or choose View|Message.

Fixing errors

To edit the code associated with an error or warning, do one of the following:

- Double-click the message in the Message window.
- Select the message in the Message window and press *Enter*.
- Press *Alt-F10* and choose Edit Source from the SpeedMenu.

The Edit window gains focus with the insertion point placed on the line and column in your source code where the compiler detected the error. From here, edit your code to fix the error. After fixing the error, press *Alt-F7* to move the next error message in the list or press *Alt-F8* to go back to the previous message.

Project options reference

You set compiler, linker, librarian, and make options from two different places in the Paradigm C++ IDE: the Project Options multiple-page dialog box and TargetExpert. The remainder of this chapter describes the options available in the Project Option dialog box. They are described in alphabetical order.

16-bit compiler options

The 16-bit compiler options affect the compilation of all 16-bit source modules. It is usually best to keep the default setting for most options in this section.

The subtopics are

- Processor
- Calling convention
- Memory model
- Segment names data
- Segment names far data
- Segment names code
- Entry/Exit code

Calling conventions

Calling Convention options tell the compiler which calling sequences to generate for function calls. The C, Pascal, and Register calling conventions differ in the way each handles stack cleanup, order of parameters, case, and prefix of global identifiers.

You can use the `__cdecl`, `__pascal`, or `__fastcall` keywords to override the default calling convention on specific functions.

C

Command-line equivalent: **-pc, -p-**

This option tells the compiler to generate a C calling sequence for function calls (generate underbars, case sensitive, push parameters right to left). This is the same as declaring all subroutines and functions with the `__cdecl` keyword. Functions declared using the C calling convention can take a variable parameter list (the number of parameters does not need to be fixed).

Pascal

Command-line equivalent: **-p**

This option tells the compiler to generate a Pascal calling sequence for function calls (do not generate underbars, all uppercase, calling function cleans stack, pushes parameters left to right). This is the same as declaring all subroutines and functions with the `__pascal` keyword. The resulting function calls are usually smaller and faster than those made with the C (**-pc**) calling convention. Functions must pass the correct number and type of arguments.

Register

Command-line equivalent: **-pr**

This option forces the compiler to generate all subroutines and all functions using the **Register** parameter-passing convention, which is equivalent to declaring all subroutine and functions with the `__fastcall` keyword. With this option enabled, functions or routines expect parameters to be passed in registers.

Default = C (**-pc**)

Memory model

The Memory Model section lets you specify the organization of segments for code and data in your 16-bit programs. All .OBJ and .LIB files in your program should be compiled in the same memory model.

The options are

- Model
- Assume SS equals DS

Options

- Put constant strings in code segments
- Far virtual tables
- Automatic far data
- Fast huge pointers
- Far data threshold

Assume SS equals DS

The Assume SS Equals DS options specify how the compiler considers the stack segment (SS) and the data segment (DS).

Default for memory model	The memory model you use determines whether the stack segment (SS) is equal to the data segment (DS). Usually, the compiler assumes that SS is equal to DS in the small and medium memory models.
---------------------------------	---

Never	Command-line equivalent: -Fs- The compiler assumes that the SS is never equal to DS. This is always the case in the compact and large memory models.
--------------	--

Always	Command-line equivalent: -Fs The compiler always assumes that SS is equal to DS in all memory models. You can use this option when porting code originally written for an implementation that makes the stack part of the data segment but you will have to provide replacement startup code for this option to work.
---------------	---

Default = Default for Memory Model

Automatic far data

Command-line equivalent: **-Ff**

When the Automatic Far Data option is enabled, the compiler automatically places data objects larger than or equal to the threshold size into far data segments. The threshold size defaults to 32,767. This option is useful for code that doesn't use the huge memory model, but declares enough large global variables that their total size is close to or exceeds 64K. This option has no effect for programs that use small, and medium memory models.

When this option is disabled, the size value is ignored

This option and the Far Data Threshold input box work together. The Far Data Threshold specifies the minimum size above that which data objects will be automatically made far.

If you use this option with the Generate COMDEFs option (**-Fc**), the COMDEFs become far in the compact, large, and huge models.

Default = OFF

The command-line option **-Fm** enables all the other **-F** options (**-Fc**, **-Ff**, and **-Fs**). You can use **-Fm** as a handy shortcut when porting code from other compilers. To do this in the Paradigm C++ IDE, check the Automatic Far Data and Always options on this Project Options page, and the Generate COMDEFs option on the Compiler|Floating Point page.

Page alignment for far segments

Command-line equivalent: **-Fa**

Allows you to change from paragraph (alignment on a 16-byte boundary) to page alignment (256-byte boundary alignment) of far segments.

Borland C++-compatible far data

Command-line equivalent: **-Fb**

Enables Borland C++ compatible far data segments. When enabled, Paradigm C++ will combine initialized and uninitialized far data into the **FAR_DATA** class instead of placing initialized far data in class **FAR_DATA** and uninitialized far data in class **FAR_BSS**.

Far data threshold

Command-line equivalent: **-Ff=size**, where *size* = threshold size

Use Far Data Threshold to specify the size portion needed to complete the Automatic Far Data option.

Default = 32767 (if Automatic Far Data is disabled, this option value is ignored)

Far virtual tables

Command-line equivalent: **-Vf**

When you turn this option on, the compiler creates virtual tables in the code segment instead of the data segment, unless you override this option using the Far Virtual Tables Segment (**-zV**) or Far Virtual Tables Class (**-zW**) options. Virtual table pointers are made into full 32-bit pointers (which is done automatically if you are using the huge memory model).

You can use Far Virtual Tables to remove the virtual tables from the data segment (which might be getting full). You might also use this option to share objects (of classes with virtual functions) between modules that use different data segments.

You must compile all modules that might share objects entirely with or entirely without this option.



You can get the same effect by using the **huge** or **_export** modifiers on a class-by-class basis.

This option changes the mangled names of C++ objects.

Default = OFF

Fast huge pointers

Command-line equivalent: **-h**

This option offers an alternative method of calculating huge pointer expressions.

For 16-bit real-mode programs, this option offers a faster method of “normalizing” than the standard method. (*Normalizing* is resolving a memory address so that the offset is always less than 16.) When you use this option, huge pointers are normalized only when a segment wraparound occurs in the offset part, which causes problems with huge arrays if an array element crosses a segment boundary.

Usually, Paradigm C++ normalizes a huge pointer whenever adding or subtracting from it. This ensures, for example, that if you have an array of **structs** that’s larger than 64K, indexing into the array and selecting a struct field always works with structs of any size. Paradigm C++ accomplishes this by always normalizing the results of huge pointer operations--the address offset contains a number that is no higher than 15 and a segment wraparound never occurs with huge pointers. The disadvantage of this approach is that it tends to be quite expensive in terms of execution speed.

Default = OFF

Model

The Model options specify the memory model you want to use. The memory model you choose determines the default method of memory addressing.

Small Command-line equivalent: **-ms**

Use the small model for average size applications. The code and data segments are different and don’t overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used.

The **-ms!** command-line option compiles using the small model and assumes DS != SS. To achieve this in the Paradigm C++ IDE, you need to check both the **Small** and **Never** options.

Medium Command-line equivalent: **-mm**

Use the medium model for large programs that do not keep much data in memory. Far pointers are used for code but not for data. Data and stack together are limited to 64K, but code can occupy up to 1 MB.

The **-mm!** command-line option compiles using the medium model and assumes DS != SS. To achieve this in the Paradigm C++ IDE, you need to check both the **Medium** and **Never** options.



The net effect of the **-ms!** and **-mm!** options is actually very small. If you take the address of a stack variable (parameter or auto), the default (DS == SS) is to make the resulting pointer a near (DS relative) pointer. This way, you can assign the address to a default-sized pointer in those models without problems. When DS != SS, the pointer type created when you take the address of a stack variable is an **_ss** pointer. This means that the pointer can be freely assigned or passed to a far pointer or to an **_ss** pointer. But for the memory models affected, assigning the address to a near or default-sized pointer produces a “Suspicious pointer conversion” warning. Such warnings are usually errors.

Compact Command-line equivalent: **-mc**

Use the compact model if your code is small but you need to address a lot of data. The Compact model is the opposite of the medium model: far pointers are used for data but not for code; code is limited to 64K, pointers can point almost anywhere. All functions are near by default and all data pointers are far by default.

Large Command-line equivalent: **-ml**

Use the large model for very large applications only. Far pointers are used for both code and data. Data is limited to 1MB. Far pointers can point almost anywhere. All functions and data pointers are far by default.

Huge Command-line equivalent: **-mh**, 16-bit real mode only

Use the huge model for very large applications only. Far pointers are used for both code and data. Paradigm C++ normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing data to occupy more than 64K.

Default = Large in the Paradigm C++ IDE; Small in PCC.EXE

Put constant strings in code segments

Command-line equivalent: **-dc**

This option moves all string literals from the data segment to the code segment of the generated object file, making the data type **const**.



Use this option only with compact or large memory models. In addition, this option does not work with overlays.

Using this option saves data segment space. In large programs, especially those with a large number of literal strings, this option shifts the burden from the data segment to the code segment.

Default = OFF

Processor

The Processor options let you specify the minimum CPU type compatible with your program. These options introduce instructions specific to the CPU type you select to increase performance.

The options are

- Instruction set
- Data alignment

16-bit instruction set

The Instruction Set options specify for which CPU instruction set the compiler should generate code.

8086 Command-line equivalent: **-1-**

Choose the 8086 option if you want the compiler to generate 16-bit code for the 8086-compatible instruction set. (To generate 8086 code, you must not turn on the options **-2**, **-3**, or **-4**, or **-5**.) This option is the default for 16-bit.

80186 Command-line equivalent: **-1**

Choose the 80186 option if you want the compiler to generate extended 16-bit code for the 80186 instruction set. Also supports the 80286 running in Real mode.

80286 Command-line equivalent: **-2**

Choose the 80286 option if you want the compiler to generate 16-bit code for the 80286 protected-mode-compatible instruction set.

80386 Command-line equivalent: **-3**

Choose the 80386 option if you want the compiler to generate 16-bit code for the 80386 protected-mode-compatible instruction set.

i486 Command-line equivalent: **-4**

Choose the i486 option if you want the compiler to generate 80386/i486 instructions running in enhanced-mode Windows.

Default = 8086 (**-1**-)

Pentium Command-line equivalent: **-5**

Choose the Pentium option if you want the compiler to generate Pentium instructions running in enhanced-mode Windows.

Data alignment

The Data Alignment options let you choose the compiler aligns data in stored memory. Word, double-word, and quad-word alignment forces integer-size and larger items to be aligned on memory addresses that are a multiple of the type chosen. Extra bytes are inserted in structures to ensure that members align correctly.

Byte alignment Command-line equivalent: **-a1** or **-a-**

When Byte Alignment is turned on, the compiler does not force alignment of variables or data fields to any specific memory boundaries; the compiler aligns data at either even or odd addresses, depending on which is the next available address.

While byte-wise alignment produces more compact programs, the programs tend to run a bit slower. The other data alignment options increase the speed that 80x86 processors fetch and store data.

Word alignment (2-byte)	<p>Command-line equivalent: -a2</p> <p>When Word Alignment is on, the compiler aligns non-character data at even addresses. Automatic and global variables are aligned properly. char and unsigned char variables and fields can be placed at any address; all others are placed at an even-numbered address.</p>
Double word (4-byte)	<p>Command-line equivalent: -a4, 32-bit only</p> <p>Double Word alignment aligns non-character data at 32-bit word (4-byte) boundaries.</p>
Quad word (8-byte)	<p>Command-line equivalent: -a8, 32-bit only</p> <p>Quad Word alignment aligns non-character data at 64-bit word (8-byte) boundaries.</p> <p>Default = Byte Alignment (-a-)</p>

Segment names code

Segment Names Code options let you specify a new code segment name and reassign the group and class.

The options are

- Code segment
- Code group
- Code class

Code

Do not change the settings in this dialog box unless you are an expert.

Use Code to change the name of the code segment as well as the code group and class.

In all options, use an asterisk (*) for **name** to select the default segment names.

Code segment	<p>Command-line equivalent = -zCname</p> <p>Sets the name of the code segment to <i>name</i>. By default, the code segment is named <code>_CODE</code> for near code and <code>modulename_TEXT</code> for far code, except for the medium and large models where the name is <code>filename_CODE</code> (<i>filename</i> is the source file name).</p>
Code group	<p>Command-line equivalent = -zPname</p> <p>Causes any output files to be generated with a code group for the code segment named <i>name</i>.</p>
Code class	<p>Command-line equivalent = -zAname</p> <p>Changes the name of the code segment class to <i>name</i>. By default, the code segment is assigned to class <code>CODE</code>.</p> <p>Default = * (default segment name) for all options</p>

Segment names data

Use Segment Names Data to change the default segment, group, and class names for initialized and uninitialized data.



Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

The options available are

- Initialized Data
- Uninitialized Data

Initialized data

Use Initialized data to change the default segment, group, and class names for initialized data.

In all options, use an asterisk (*) for **name** to select the default segment names.



Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

Initialized data class

Command-line equivalent = **-zTname**

Sets the name of the initialized data segment to *name*. By default, the initialized data segments class is named DATA.

Default = * (default segment name) for all options

Initialized data group

Command-line equivalent = **-zSname**

Sets the name of the initialized data segment group to *name*. By default, the data group is named DGROUP.

Initialized data segment

Command-line equivalent = **-zRname**

Sets the name of the initialized data segment to *name*. By default, the initialized data segment is named _DATA for **near** data and *modulename*_DATA for **far** data.


Uninitialized data

Use Uninitialized Data to change the default segment, group, and class names for code uninitialized data.

In all options, use an asterisk (*) for **name** to select the default segment names.



Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

Uninitialized data (BSS class)	<p>Command-line equivalent = -zBname</p> <p>Sets the name of the uninitialized data segment class to <i>name</i>. By default, the uninitialized data segments are assigned to class BSS.</p> <p>Default = * (default segment name) for all options</p>
Uninitialized data (BSS group)	<p>Command-line equivalent = -zGname</p> <p>Sets the name of the uninitialized data segment group to <i>name</i>. By default, the data group is named DGROUP.</p>
Uninitialized data (BSS segment)	<p>Command-line equivalent = -zDname</p> <p>Sets the name of the uninitialized data segment. By default, the uninitialized data segment is named <code>_BSS</code> for near uninitialized data and <i>modulename</i>_BSS for far uninitialized data.</p> <p>Segment names far data</p> <p>16-bit Compiler Segment Names Far Data options set the far data segment name, group, class name, and the far virtual tables segment name and class.</p> <p>Far initialized data</p> <p>Use the far uninitialized data options to change the default segment, group, and class names for far initialized data. These options also apply to far uninitialized data if the -Fb option is enabled. In all options, use an asterisk (*) for <i>name</i> to select the default segment names.</p> <p> Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.</p>
Far data class	<p>Command-line equivalent = -zFname</p> <p>Sets the name of the class for <code>__far</code> initialized objects to <i>name</i>. By default, the name is FAR_DATA.</p> <p>Default = * (default segment name) for all options</p>
Far data group	<p>Command-line equivalent = -zHname</p> <p>Causes <code>__far</code> initialized objects to be placed into the group <i>name</i>. By default, far objects are not placed into a group.</p>
Far data segment	<p>Command-line equivalent = -zEname</p> <p>Sets the name of the segment where <code>__far</code> initialized objects are placed to <i>name</i>. By default, the segment name is the name of the object module followed by <code>_DATA</code>.</p>

Far uninitialized data

Use the far uninitialized data options to change the default segment, group, and class names for far uninitialized data. In all options, use an asterisk (*) for *name* to select the default segment names.



Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

Far uninitialized data class

Command-line equivalent = **-zYname**

Sets the name of the class for **__far** uninitialized objects to *name*. By default, the name is FAR_BSS.

Default = * (default segment name) for all options

Far uninitialized data group

Command-line equivalent = **-zZname**

Causes uninitialized **__far** objects to be placed into the group *name*. By default, far uninitialized objects are not placed into a group.

Far uninitialized data segment

Command-line equivalent = **-zXname**

Sets the name of the segment where uninitialized **__far** objects are placed to *name*. By default, the segment name is the name of the object module followed by _BSS.

Far virtual tables

Use Far Virtual Tables to change the default segment and class names virtual tables.

In all options, use an asterisk (*) for *name* to select the default segment names.



Do not change the settings in this dialog box unless you have a good understanding of segmentation on the 80x86 processor. Under normal circumstances, you do not need to specify segment names.

Virtual table class

Command-line equivalent = **-zWname**

Sets the name of the far virtual table class segment to *name*. By default, far virtual table classes are generated in the CODE segment.

Default = * (default segment name) for all options

Virtual table segment

Command-line equivalent = **-zVname**

Sets the name of the **__far** virtual table segment to *name*. By default, far virtual tables are generated in the CODE segment.

32-bit compiler options

The 32-bit Compiler page contains two radio buttons that allow you to select which 32-bit compiler backend you want to use when compiling 32-bit applications.

Paradigm optimizing compiler

The Paradigm optimizing compiler is a faster compiler than the Intel compiler, and it produces smaller executable files. If you are compiling from the command line, use PCC32.EXE.

Intel optimizing compiler

The Intel optimizing compiler produces faster executable files than the Paradigm compiler. The trade-off is slower compilation times and slightly larger executable file sizes. If you are compiling from the command line, use PCC32i.EXE.



The Intel compiler does not support the Browser Information (-R) compiler option.

32-bit compiler options

32-bit compiler options listed on the Processor and Calling Convention pages affect the compilation of all 32-bit Windows applications for Windows NT and Windows 95. Because 32-bit programs use a flat memory model (they are not segmented), there are fewer options to configure than for 16-bit programs.

Calling conventions

Calling Convention options tell the compiler which calling sequences to generate for function calls. The C, Pascal, and Register calling conventions differ in the way each handles stack cleanup, order of parameters, case, and prefix of global identifiers.

These options should be used by experts only.

You can use the `_cdecl`, `_pascal`, `_fastcall`, or `_stdcall` keywords to override the default calling convention on specific functions.

C Command-line equivalent: **-pc, -p-**

This option tells the compiler to generate a C calling sequence for function calls (generate underbars, case sensitive, push parameters right to left). This is the same as declaring all subroutines and functions with the `_cdecl` keyword. Functions declared using the C calling convention can take a variable parameter list (the number of parameters does not need to be fixed).

You can use the `_pascal`, `_fastcall`, or `_stdcall` keywords to specifically declare a function or subroutine using another calling convention.

Pascal Command-line equivalent: **-p**

This option tells the compiler to generate a Pascal calling sequence for function calls (do not generate underbars, all uppercase, calling function cleans stack, pushes parameters left to right). This is the same as declaring all subroutines and functions with the `_pascal` keyword. The resulting function calls are usually smaller and faster than those made with the C (**-pc**) calling convention. Functions must pass the correct number and type of arguments.

You can use the `_cdecl`, `_fastcall`, or `_stdcall` keywords to specifically declare a function or subroutine using another calling convention.

Register Command-line equivalent: **-pr**

This option forces the compiler to generate all subroutines and all functions using the **Register** parameter-passing convention, which is equivalent to declaring all subroutine and functions with the **__fastcall** keyword. With this option enabled, functions or routines expect parameters to be passed in registers.

You can use the **__pascal**, **__cdecl**, or **__stdcall** keywords to specifically declare a function or subroutine using another calling convention.

Standard Call Command-line equivalent: **-ps**

This option tells the compiler to generate a Stdcall calling sequence for function calls (does not generate underscores, preserve case, called function pops the stack, and pushes parameters right to left). This is the same as declaring all subroutines and functions with the **__stdcall** keyword. Functions must pass the correct number and type of arguments.

You can use the **__cdecl**, **__pascal**, **__fastcall** keywords to specifically declare a function or subroutine using another calling convention.

Default = C (**-pc**)

Processor

The 32-bit Compiler Processor options specify which CPU instruction set to use and how to handle floating-point code for 32-bit programs.

32-bit instruction set The Instruction Set options specify for which CPU instruction set the compiler should generate code.

80386 Command-line equivalent: **-3**

Choose the 80386 option if you want the compiler to generate 80386 protected-mode compatible instructions running on Windows 95 or Windows NT.

i486 Command-line equivalent: **-4**

Choose the i486 option if you want the compiler to generate i486 protected-mode compatible instructions running on Windows 95 or Windows NT.

Pentium Command-line equivalent: **-5**

Choose the Pentium option if you want the compiler to generate Pentium instructions on Windows 95 or Windows NT.

While this option increases the speed at which the application runs on Pentium machines, expect the program to be a bit larger than when compiled with the **80386** or **i486** options. In addition, **Pentium**-compiled code will sustain a performance hit on non-Pentium systems.

Default = 80386 (**-3**)

Build attributes

Build attributes affect whether or not a node is built during compilation. The icons associated with each of these options are displayed next to the nodes in the Project hierarchy diagram. Build attributes are set in the Options|Project dialog box.

Always build

Check Always Build and the node is always built, even if it has not changed.

Build when out of date

Check Build When Out of Date and the node is built only if it has changed.

Never build

Check Never Build and the node is not built.

Can't build

Check Can't Build to be notified when a node cannot be built.

Exclude from parent

Check Exclude from Parent and the system indicates when a node should be excluded from parent (such as with source pools).

C++ options

Project|C++ Options affect compilation of all C and C++ programs. For most of the C++ options, you'll usually want to use the default settings.

C++ compatibility

Use the C++ Compatibility options to handle C++ compatibility issues, such as handling 'char' types, specifying options about hidden pointers, passing class arguments, adding hidden members and code to a derived class, passing the 'this' pointer to 'Pascal' member functions, changing the layout of classes, or insuring compatibility when class instances are shared with non-C++ code or code compiled with previous versions of Paradigm C++.

'deep' virtual bases

(Command-line equivalent: **-Vv**)

When a derived class overrides a virtual function which it inherits from a virtual base class, and a constructor or destructor for the derived class calls that virtual function using a pointer to the virtual base class, the compiler can sometimes add hidden members to the derived class. These “hidden members” add code to the constructors and destructors.

This option directs the compiler *not* to add the hidden members and code so that the class instance layout is the same as with previous version of Paradigm C++; the compiler does not change the layout of any classes to relax the restrictions on pointers.

Default = OFF

Calling convention mangling compatibility

(Command-line equivalent: **-VC**)

When this option is enabled, the compiler disables the distinction of function names where the only possible difference is incompatible code generation options. For example, with this option enabled, the linker will not detect if a call is made to a `__fastcall` member function with the `cdecl` calling convention.

This option is provided for backward compatibility only; it lets you link old library files that you cannot recompile.

Default = OFF

Disable constructor displacements

(Command-line equivalent: **-Vc**)

When the Disable Constructor Displacements option is enabled, the compiler does not add hidden members and code to a derived class (the default).

This option insures compatibility with previous versions of the compiler.

Default = OFF

Do not treat 'char' as distinct type

(Command-line equivalent: **-K2**, 16-bit)

Allow only *signed* and *unsigned char* types. The Paradigm C++ compiler allows for signed char, unsigned char, and char data types. This option treats *char* as signed.

This option is provided for compatibility with previous versions of Paradigm C++ (3.1 and earlier) and supports only 16-bit programs.

Default = OFF

Don't restrict scope of 'for' loop expression variables

Command-line equivalent: **-Vd**

This option lets you specify the scope of variables declared in **for** loop expressions. The output of the following code segment changes, depending on the setting of this option.

```
int main(void)
{
    for(int i=0; i<10; i++)
    {
        cout << "Inside for loop, i = " << i << endl;
    }    //end of for-loop block

    cout << "Outside for loop, i = " << i << endl; //error without
                                                -Vd
}    //end of block containing for loop
```

If this option is disabled (the default), the variable *i* goes out of scope when processing reaches the end of the **for** loop. Because of this, you'll get an Undefined Symbol compilation error if you compile this code with this option disabled.

If this option is enabled (**-Vd**), the variable *i* goes out of scope when processing reaches the end of the block containing the **for** loop. In this case, the code output would be:

```
Inside for loop, i = 0
...
Outside for loop, i = 10
```

Default = OFF

Pass class values via reference to temporary

Command-line equivalent: **-Va**

*This option
insures
compatibility with
previous
versions of the
compiler.*

When this option is enabled, the compiler passes class arguments using the "reference to temporary" approach. When an argument of type class with constructors is passed by value to a function, this option instructs the compiler to create a temporary variable at the calling site, initialize this temporary variable with the argument value, and pass a reference from this temporary to the function.

Default = OFF

Push 'this' first for Pascal member functions

Command-line equivalent: **-Vp**

When this option is enabled, the compiler passes the **this** pointer to Pascal member functions as the first parameter on the stack.

By default, the compiler passes the **this** parameter as the last parameter on the stack, which permits smaller and faster member function calls.

Default = OFF

Treat 'far' classes as 'huge'

Command-line equivalent **-Vh**

When this option is enabled, the compiler treats all classes declared **__far** as if they were declared as **__huge**. For example, the following code normally fails to compile. Checking this option allows the following code fragment to compile:

```
struct __huge A
{
    virtual void f(); // A vtable is required to see the error.
};
struct __far B : public A
{
};
// Error: Attempting to derive a far class from the huge base 'A'.
```

Default = OFF

Virtual base pointers

*The Virtual Base
Pointers options
specify options
about the hidden
pointer.*

When a class inherits virtually from a base class, the compiler stores a hidden pointer in the class object to access the virtual base class subobject.

Always near Command-line equivalent: **-Vb-**

This option allows for the smallest and most efficient code. When the Always Near option is on, the hidden pointer will always be a near pointer. (When a class inherits virtually from a base class, the compiler stores a *hidden pointer* in the class object to access the virtual base class subobject.)

Same size as 'this' pointer Command-line equivalent: **-Vb**

When the Same Size as 'this' Pointer option is on, the compiler matches the size of the hidden pointer to the size of the **this** pointer in the instance class.

This allows for compatibility with previous versions of the compiler.

Default = Always Near (**-Vb-**)

Vtable pointer follows data members

Command-line equivalent **-Vt**

When this option is enabled, the compiler places the virtual table pointer after any nonstatic data members of the specified class.

This option insures compatibility when class instances are shared with non-C++ code and when sharing classes with code compiled with previous versions of Paradigm C++.

Default = OFF

Exception handling/RTTI

Use the Exceptions Handling options to enable or disable exception handling and to tell the compiler how to handle the generation of run-time type information.

If you use exception handling constructs in your code and compile with exceptions disabled, you'll get an error.

Enable exceptions

Command-line equivalent: **-x**

When this option is enabled, C++ exception handling is enabled. If this option is disabled (**-x-**) and you attempt to use exception handling routines in your code, the compiler generates error messages during compilation.

Disabling this option makes it easier for you to remove exception handling information from programs; this might be useful if you are porting your code to other platforms or compilers.



Disabling this option turns off only the compilation of exception handling code; your application can still include exception code if you link .OBJ and library files that were built with exceptions enabled (such as the Paradigm standard libraries).

Default = ON

Enable run-time type information

Command-line equivalent: **-RT**

This option causes the compiler to generate code that allows run-time type identification.

In general, if you set Enable Destructor Cleanup (**-xd**), you will need to set this option as well.

Default = ON

**Enable
exception
location
information**

Command-line equivalent: **-xp**

When this option is enabled, run-time identification of exceptions is available because the compiler provides the file name and source-code line number where the exception occurred. This enables the program to query file and line number from where a C++ exception was thrown.

Default = OFF

**Enable
destructor
cleanup**

Command-line equivalent: **-xd**

When this option is enabled and an exception is thrown, destructors are called for all automatically declared objects between the scope of the *catch* and *throw* statements.

In general, when you enable this option, you should also set Enable Runtime Type Information (**-RT**) as well.



Destructors are not automatically called for dynamic objects allocated with **new**, and dynamic objects are not automatically freed.

Default = ON

**Enable fast
exception
prologs**

Command-line equivalent: **-xf**

When this option is enabled, inline code is expanded for every exception handling function. This option improves performance at the cost of larger executable file sizes.



If you select both Fast Exception Prologs and Enable Compatible Exceptions (**-xc**), fast prologs will be generated but Enable Compatible Exceptions will be disabled (the two options are not compatible).

Default = OFF

**Enable
compatible
exceptions**

Command-line equivalent: **-xc**, 16-bit only

This option allows .AXEs and .DLLs built with Paradigm C++ to be compatible with executables built with other products. When Enable Compatible Exceptions is disabled, some exception handling information is included in the .AXE, which could cause compatibility issues.



Libraries that can be linked into .DLLs need to be built with this option enabled.

Default = OFF

General

Zero-length empty base classes

Command-line equivalent: **-Ve**

Usually the size of a class is at least one byte, even if the class does not define any data members. When this option is enabled, the compiler ignores this unused byte for the memory layout and the total size of any derived classes.

Default = OFF

Member pointer

Use C++ Member Pointers options to direct member pointers and affect how the compiler treats explicit casts.

Honor precision of member pointers

Command-line equivalent: **-Vmp**

When this option is enabled, the compiler uses the declared precision for member pointer types. Use this option when a pointer to a derived class is explicitly cast as a pointer-to-member of a simpler base class (when the pointer is actually pointing to a derived class member).

Default = OFF

Member pointer representation

The C++ Member pointers options specify what member pointers can point to.

Support all cases

Command-line equivalent: **-Vmv**

When this option is enabled, the compiler places no restrictions on where member pointers can point. Member pointers use the most general (but not always the most efficient) representation.

Default = ON

Support multiple inheritance

Command-line equivalent: **-Vmm**

When this option is enabled, member pointers can point to members of multiple inheritance classes (with the exception of virtual base classes).

Default = OFF

Support single inheritance

Command-line equivalent: **-Vms**

When this option is enabled, member pointers can point only to members of base classes that use single inheritance.

Default = OFF

Smallest for class

Command-line equivalent: **-Vmd**

When this option is enabled, member pointers use the smallest possible representation that allows member pointers to point to all members of their particular class. If the class is not fully defined at the point where the member pointer type is declared, the most general representation is chosen by the compiler and a warning is issued.

Default = OFF

Templates

Use the options under C++ Options|Templates to tell the compiler how to generate template instances in C++.

Templates instance generation

The Template Instance Generation options specify how the compiler generates template instances in C++.

Smart	Command-line equivalent: -Jg When the Smart option is enabled, the compiler generates public (global) definitions for all template instances. If more than one module generates the same template instance, the linker automatically merges duplicates to produce a single copy of the instance. <i>This is a convenient way of generating template instances.</i> To generate the instances, the compiler must have available the function body (in the case of a template function) or the bodies of member functions and definitions for static data members (in the case of a template class), typically in a header file. Default = ON
--------------	---

Global	Command-line equivalent: -Jgd When the Global option is on, the compiler generates public (global) definitions for all template instances. The Global option does not merge duplicates. If the same template instance is generated more than once, the linker reports public symbol re-definition errors. Default = OFF
---------------	---

External	Command-line equivalent: -Jgx When the External option is on, the compiler generates external references to all template instances. When you use this option, all template instances in your code must be publicly defined in another module with the external option (-Jgd) so that external references are properly resolved. Default = OFF
-----------------	--

Virtual tables

C++ Options|Virtual Tables options control C++ virtual tables and the expansion of inline functions when debugging.

Virtual tables linkage

The C++ Virtual Tables options control C++ virtual tables and the expansion of inline functions when debugging.

Smart	Command-line equivalent: -V
--------------	------------------------------------

This option generates common C++ virtual tables and out-of-line inline functions across the modules in your application. As a result, only one instance of a given virtual table or out-of-line inline function is included in the program.

The Smart option generates the smallest and most efficient executables, but produces .OBJ and .ASM files compatible only with PLINK and PASM.

Default = ON

Local Command-line equivalent: **-Vs**

You use the Local option to generate local virtual tables (and out-of-line inline functions) so that each module gets its own private copy of each virtual table or inline function it uses.

The Local option uses only standard .OBJ and .ASM constructs, but produces larger executables.

Default = OFF

External Command-line equivalent: **-V0**

You use the External option to generate external references to virtual tables. If you don't want to use the Smart or Local options, use the External and Public options to produce and reference global virtual tables.



When you use this option, one or more of the modules comprising the program must be compiled with the Public option to supply the definitions for the virtual tables.

Default = OFF

Public Command-line equivalent: **-V1**

Public produces public definitions for virtual tables. When using the External option (-V0), at least one of the modules in the program must be compiled with the Public option to supply the definitions for the virtual tables. All other modules should be compiled with the External option to refer to that Public copy of the virtual tables.

Default = OFF

Compiler options

Compiler options are common to all C and C++ programs. They directly affect how the compiler generates code.

Defines

Command-line equivalent: **-Dname** and **-Dname=string**

The macro definition capability of Paradigm C++ lets you define and undefine macros (also called *manifest* or *symbolic* constants) in the Paradigm C++ IDE or on the command line. The macros you define override those defined in your source files.



You can use the **\$INHERIT** and **\$ENV()** macros to specify the defines for the project node you are modifying.

Defining macros from the Paradigm C++ IDE

Preprocessor definitions (such as those used in #if statements and macro definitions) can be entered on the Compiler Defines page. The following rules apply when using the Defines input box:

- Separate multiple definitions with semicolons (;), and assign values with an equal sign (=). For example:

```
Switch1;Switch2;Switch3=OFF
```

- Leading and trailing spaces are stripped, but embedded spaces are left intact.
- If you want to include a semicolon in a macro, precede the semicolon with a backslash (\).

Defining macros on the command line

On the command line, the **-Dname** option defines the identifier *name* to the null string. **-Dname=string** defines *name* to *string*. In this assignment, *string* cannot contain spaces or tabs. You can also define multiple **#define** options on the command line using either of the following methods:

- Include multiple definitions after a single **-D** option by separating each define with a semicolon (;) and assigning values with an equal sign (=). For example:

```
PCC.EXE -Dxxx;yyy=1;zzz=NO MYFILE.C
```

- Include multiple **-D** options, separating each with a space. For example:

```
PCC.EXE -Dxxx -Dyyy=1 -Dzzz=NO MYFILE.C
```

Code generation

Compiler Code Generation options affect how code is generated.

Allocate enums as ints

Command-line equivalent: **-b**

When the Allocate Enums As Ints option is on, the compiler always allocates a whole word (a two-byte **int** for 16-bits or a four-byte **int** for 32-bits) for enumeration types (variables of type **enum**).

When this option is off (**-b-**), the compiler allocates the smallest integer that can hold the enumeration values: the compiler allocates an **unsigned** or **signed char** if the values of the enumeration are within the range of 0 to 255 (minimum) or -128 to 127 (maximum), or an **unsigned** or **signed short** if the values of the enumeration are within the following ranges:

- 0 to 65,535 (minimum) or -32,768 to 32,767 (maximum) (16-bit)
- 0 to 4,294,967,295 or -2,147,483,648 to 2,147,483,647 (32-bit)

The compiler allocates a two-byte **int** (16-bit) or a four-byte **int** (32-bit) to represent the enumeration values if any value is out of range.

Default = ON

Duplicate strings merged

Command-line equivalent: **-d**

When you check the Duplicate Strings Merged option, the compiler merges two literal strings when one matches another. This produces smaller programs (at the expense of a slightly longer compile time), but can introduce errors if you modify one string.

Default = OFF (**-d-**)

fastthis

Command-line equivalent: **-po**, 16-bit only

This option causes the compiler to use the `__fastthis` calling convention when passing the **this** pointer to member functions. The **this** pointer is passed in a register (or a register pair in 16-bit large data models). Likewise, calls to member functions load the register (or register pair) with **this**. Note that you can use `__fastthis` to compile specific functions in this manner.

When **this** is a 'near' (16-bit) pointer, it is supplied in the SI register; for 'far' **this** pointers, DS:SI is used. If necessary, the compiler saves and restores DS. All references in the member function to member data are done via the SI register.

The names of member functions compiled with `__fastthis` are mangled differently from non-fastthis member functions, to prevent mixing the two. It is easiest to compile all classes with `__fastthis`, but you can compile some classes with `__fastthis` and some without, as in the following example:

```
// no -po on the command-line
class X;
#pragma option -po
class Y      //Y will use fastthis
{
    ...
};
class X      //X will not use fastthis,
{           //since its class declaration
           //appeared before fastthis was turned on
    ...
};
#pragma option -po-
```



If you use a makefile to build a version of the class library that has `__fastthis` enabled, you must define `CLASSLIB_ALLOW_po` and use the **-po** option. The `__CLASSLIB_ALLOW_po` macro can be defined in `<Your_PCW_dir>\INCLUDE\paradigm.h`

If you use a makefile to build a `__fastthis` version of the run-time library, you must define `__RTL_ALLOW_po` and use the **-po** option.

If you rebuild the libraries and use **-po** without defining the appropriate macro, the linker emits undefined symbol errors.

Default = OFF

Register variables

These options suppress or enable the use of register variables.

None Command-line equivalent: **-r-**

Choose None to tell the compiler not to use register variables even if you have used the **register** keyword.

Register keyword

You can use **-rd** in **#pragma options**.

Command-line equivalent: **-rd**

Choose Register Keyword to tell the compiler to use register variables only if you use the **register** keyword and a register is available. Use this option or the Automatic option (**-r**) to optimize the use of registers.

Automatic

Command-line equivalent: **-r**

Choose Automatic to tell the compiler to automatically assign register variables if possible, even when you do not specify a register variable by using the **register** type specifier.

Generally, you can keep this option set to Automatic unless you are interfacing with preexisting assembly code that does not support register variables.

Default = Automatic (**-r**)

Unsigned characters

Command-line equivalent: **-K**

When the Unsigned Characters option is on, the compiler treats all **char** declarations as if they were **unsigned char** type, which provides compatibility with other compilers.

Default = OFF (**char** declarations default to **signed**; **-K-**)

Floating point

The Floating Point options specify how the compiler handles floating-point numbers in your code.

Correct Pentium FDIV flaw

Command-line equivalent: **-fp**

Some early Pentium chips do not perform specific floating-point division calculations with full precision. Although your chances of encountering this problem are slim, this switch inserts code that emulates floating-point division so that you are assured of the correct result. This option decreases your program's FDIV instruction performance.



Use of this option only corrects FDIV instructions in modules that you compile. The run-time library also contains FDIV instructions which are not modified by the use of this switch. To correct the run-time libraries, you must recompile them using this switch.

The following functions use FDIV instructions in assembly language which are not corrected if you use this option:

acos	cosh	pow10l
acosl	coshl	powl
asin	cosl	sin
asinx	exp	sinh
atan	expl	sinhl

atan2	fmod	sinl
atan2l	fmodl	tan
atanl	pow	tanh
cos	pow10	tanh1
tanl		

In addition, this switch does not correct functions that convert a floating-point number to or from a string (such as *printf* or *scanf*).

Default = OFF

No floating point

Command-line equivalent: **-f-**

Choose No Floating Point if you are not using floating point. No floating-point libraries are linked when this option is enabled (**-f-**). If you enable this option and use floating-point calculations in your program, you will get link errors. When unchecked (**-f**), the compiler emulates 80x87 calls at run-time.

Default = OFF (**-f**)

Fast floating point

Command-line equivalent: **-ff**

When Fast Floating Point is on, floating-point operations are optimized without regard to explicit or implicit type conversions. Calculations can be faster than under ANSI operating mode.

When this option is unchecked (**-ff-**), the compiler follows strict ANSI rules regarding floating-point conversions.

Default = OFF

Compiler output

Set control of object file contents on the Compiler Output page.

Autodependency information

Command-line equivalent: **-X-**

When the Autodependency option is checked (**-X-**), the compiler generates autodependency information for all project files with a .C or .CPP extension.

The Project Manager can use autodependency information to speed up compilation times. The Project Manager opens the .OBJ file and looks for information about files included in the source code. This information is always placed in the .OBJ file when the source module is compiled. After that, the time and date of every file that was used to build the .OBJ file is checked against the time and date information in the .OBJ file. The source file is recompiled if the dates are different. This is called an autodependency check.

If the project file contains valid dependency information, the Project Manager does the autodependency check using that information. This is much faster than reading each .OBJ file.

When this option is unchecked (**-X**), the compiler does not generate the autodependency information.

Modules compiled with autodependency information can use MAKE's autodependency feature.

Default = ON (**-X-**)

Generate COMDEFs

Command-line equivalent: **-Fc**, 16-bit only

Generate COMDEFs generates communal variables (COMDEFs) for global C variables that are not initialized and not declared as **static** or **extern**. Use this option when header files included in several source files contain global variables.

For example, a definition such as

```
int SomeArray[256];
```

could appear in a header file that is then included in many modules. When this option is on, the compiler generates *SomeArray* as a communal variable rather than a public definition (a COMDEF record rather than a PUBDEF record). You can use this option when porting code that uses a similar feature with another implementation.

The linker generates only one instance of the variable, so it will not be a duplicate definition linker error. As long as a given variable does not need to be initialized to a nonzero value, you do not need to include a definition for it in any of the source files.

Default = OFF

Generate underscores

Command-line equivalent: **-u**

When the Generate Underscores option is on, the compiler automatically adds an underscore character (**_**) in front of every global identifier (functions and global variables) before saving them in the object module. Pascal identifiers (those modified by the **__pascal** keyword) are converted to uppercase and are not prefixed with an underscore.

Underscores for C and C++ are optional, but you should turn this option on to avoid errors if you are linking with the standard Paradigm C++ libraries.

Default = ON

Source

Compiler|Source options set source code interpretation.

Identifier length

Command-line equivalent: **-in**, where *n* = significant characters

Use the Identifier Length input box to specify the number of significant characters (those which will be recognized by the compiler) in an identifier.

Except in C++, which recognizes identifiers of unlimited length, all identifiers are treated as distinct only if their significant characters are distinct. This includes variables, preprocessor macro names, and structure member names.

Valid numbers for *n* are 0, and 8 to 250, where 0 means use the maximum identifier length of 250.

By default, Paradigm C++ uses 250 characters per identifier. Other systems (including some UNIX compilers) ignore characters beyond the first eight. If you are porting to other environments, you might want to compile your code with a smaller number of significant characters, which helps you locate name conflicts in long identifiers that have been truncated.

Default = 250

Language compliance

The Language Compliance options tell the compiler how to recognize keywords in your programs.

Paradigm extensions

Command-line equivalents: **-A-**, **-AT**

The Paradigm Extensions option tells the compiler to recognize Paradigm's extensions to the C language keywords, including **near**, **far**, **huge**, **asm**, **cdecl**, **pascal**, **interrupt**, **_export**, **_ds**, **_cs**, **_ss**, **_es**, and the register pseudovariables (**_AX**, **_BX**, and so on). For a complete list of keywords, see the *keyword index*.

ANSI

Command-line equivalent: **-A**

The ANSI option compiles C and C++ ANSI-compatible code, allowing for maximum portability. Non-ANSI keywords are ignored as keywords.

UNIX V

Command-line equivalent: **-AU**

The UNIX V option tells the compiler to recognize only UNIX V keywords and treat any of Paradigm's C++ extension keywords as normal identifiers.

Kernighan and Ritchie

Command-line equivalent: **-AK**

The Kernighan and Ritchie option tells the compiler to recognize only the K&R extension keywords and treat any of Paradigm's C++ extension keywords as normal identifiers.



If you get declaration syntax errors from your source code, check that this option is set to Paradigm Extensions.

Default = Paradigm Extensions (**-A-**)

- Accepts and ignores directives

Nested comments

Command-line equivalent: **-C**

When the Nested Comments option is on, you can nest comments in your C and C++ source files.

Nested comments are not allowed in standard C implementations, and they are not portable.

Default = OFF

Debugging

Compiler Debugging options affect the generation of debug information during compilation. When linking larger .OBJ files, you may need to turn these options off to increase the available system resources.

Browser reference information in OBJs

Command-line equivalent: **-R**

When the Browser Info In OBJs option is on, the compiler generates additional browser-specific information such as location and reference information. This information is then included in your .OBJ files. In addition to this option, you need debugging information (**-v**) to use the Browser.

When this option is off, you can link and create larger object files. While this option does not affect execution speed, it does affect compilation time and program size.

Default = ON

Debug information in OBJs

Command-line equivalent: **-v**

When the Debug Info In OBJs option is on, debugging information is included in your .OBJ files. The compiler passes this option to the linker so it can include the debugging information in the .AXE file. For debugging, this option treats C++ *inline functions* as normal functions.

You need debugging information to use either the integrated debugger or Paradigm DEBUG.

When this option is off (**-v-**), you can link and create larger object files. While this option does not affect execution speed, it does affect compilation and link time.



When Line Numbers is on, make sure you turn off Jump Optimization in the 16-bit specific optimizations and Pentium scheduling in the 32-bit Compiler options. When these options are enabled, When this option is enabled, the source code will not exactly match the generated machine instructions, which can make stepping through code confusing.

Default = ON

Line numbers

Command-line equivalent: **-y**

When the Line Numbers option is on, the compiler automatically includes line numbers in the object and object map files. Line numbers are used by both the Paradigm C++ IDE and by Paradigm DEBUG.

Although the Debug Info in OBJs option (**-v**) automatically generates line number information, you can turn that option off (**-v-**) and turn on Line Numbers (**-y**) to reduce the size of the debug information generated. With this setup, you can still step, but you will not be able to watch or inspect data items.

Including line numbers increases the size of the object and map files but does not affect the speed of the executable program.



When Line Numbers is on, make sure you turn off Jump Optimization in the 16-bit specific optimizations and Pentium scheduling in the 32-bit Compiler options. When these

options are enabled, When this option is enabled, the source code will not exactly match the generated machine instructions, which can make stepping through code confusing.

Default = OFF

Out-of-line inline functions

Command-line equivalent: **-vi**

When the Out-of-line inline functions option is on, the compiler expands C++ inline functions inline.

To control the expansion of inline functions, the Debug information in OBJs option (**-v**) acts slightly different for C++ code: when inline function expansion is disabled, inline functions are generated and called like any other function.

Because debugging with inline expansion can be difficult, the command-line compilers provide the following options:

- **-v** turns debugging on and inline expansion off
- **-v-** turns debugging off and inline expansion on
- **-vi** turns inline function expansion on
- **-vi-** turns inline expansion off (inline functions are expanded out of line)

For example, if you want to turn both debugging and inline expansion on, use the **-v** and **-vi** options.

Default = OFF

Standard stack frame

Command-line equivalent: **-k**

When the Standard stack frame option is on, the compiler generates a standard stack frame (standard function entry and exit code). This is helpful when debugging, since it simplifies the process of stepping through the stack of called subroutines.

When this option is off, any function that does not use local variables and has no parameters is compiled with abbreviated entry and return code. This makes the code smaller and faster.

The Standard stack frame option should always be on when you compile a source file for debugging.

Default = ON

Test stack overflow

Command-line equivalent: **-N**, 16-bit only

When this option is on, the compiler generates stack overflow logic at the entry of each function.

Even though this is costly in terms of both program size and speed, it can be a real help when trying to track down difficult stack overflow bugs. If an overflow is detected, the run-time error message `Stack overflow!` is generated, and the program exits with an exit code of 1.

Stack overflow testing is always enabled in the 32-bit compilers (this adds a minimal overhead to 32-bit programs). (add note sidebar)

Default = OFF

Precompiled headers

Using precompiled header files can dramatically increase compilation speed by storing an image of the symbol table on disk in a file, then later reloading that file from disk instead of parsing all the header files again. Directly loading the symbol table from disk is much faster than parsing the text of header files, especially if several source files include the same header file.



You can use the **\$INHERIT** and **\$ENV()** macros in any of the precompiled header input fields.

Cache precompiled header

Command-line equivalent: **-Hc**

When you enable this option, the compiler caches the precompiled headers it generates. This is useful when you are precompiling more than one header file.



To use this option, you must also enable the Generate and Use (**-H**) precompiled header option.

Default = OFF

Precompiled header name

Command-line equivalent: **-H=filename**

This option lets you specify the name of your precompiled header file. The compilers set the name of the precompiled header to *filename*.

When this option is enabled, the compilers generate and use the precompiled header file that you specify.

Precompiled headers

Using precompiled headers can dramatically increase compilation speeds, though they require a considerable amount of disk space.

Generate and use

Command-line equivalent: **-H**

When this option is enabled, the Paradigm C++ IDE generates and uses precompiled headers. The default file name is *<projectname>.CSM* for the Paradigm C++ IDE projects and *PCDEF.CSM* (16-bit) or *PC32DEF.CSM* (32-bit) for the command-line compilers.

Use but do not generate

Command-line equivalent: **-Hu**

When the Use But Do Not Generate option is on, the compilers use preexisting precompiled header files; new precompiled header files are not generated.

Do not generate or use

Command-line equivalent: **-H-**

When the Do not generate or use option is on, the compilers do not generate or use precompiled headers.

Default = Do not generate or use (**-H-**)

Stop precompiling after header file

Command-line equivalent: **-H"xxx"**; for example **-H"stdio.h"**

This option terminates compiling the precompiled header after the compiler compiles the file specified as *xxx*. You can use this option to reduce the amount of disk space used by precompiled headers.

When you use this option, the file you specify must be included from a source file for the compiler to generate a .CSM file.



You cannot specify a header file that is included from another header file. For example, you cannot list a header included by windows.h because this would cause the precompiled header file to be closed before the compilation of windows.h was completed.

Directories options

The Directories options tell the Paradigm C++ compiler where to find or where to put header files, library files, source code, output files, and other program elements.

Source directories

The Source directories options let you specify the directories that contain your standard include files, library and .OBJ files, and program source files.

Click the down-arrow icon or press *Alt+Down* arrow to display the history list of previously entered directory names.



You can use the **\$INHERIT** and **\$ENV()** macros in any of the following input fields.

Include

Command-line equivalent: **-Ipath**, where *path* = directory path

Use the Include list box to specify the drive and/or directories that contain program include files. Standard include files are those given in angle brackets (<>) in an **#include** statement (for example, **#include <myfile>**).



The Paradigm compilers and linkers use specific *file search algorithms* to locate the files needed to complete the compilation and link cycles.

Library

Command-line equivalent: **-Lpath**, where *path* = directory path

Use the Library list box to specify the directories that contain the Paradigm C++ startup object files (C0x.OBJ), run-time library files (.LIB files), and all other .LIB files. By default, the linker looks for them in the directory containing the project file (or in the current directory if you're using the command-line compiler).



You can also use the linker option **/Lpath** to specify the library search directories when you link files from the command line.

Source

The Source list box specifies the directories where the compiler and the integrated debugger should look for your project source files.

Specifying multiple directories

Multiple directory names are allowed in each of the list boxes; use a semicolon (;) to separate the specified drives and directories. To display a history list of previously entered directory names, click the down-arrow icon or press *Alt+Down* arrow.

From the command line, you can enter multiple include and library directories in the following ways:

- You can stack multiple entries with a single **-L** or **-I** option by separating directories with a semicolon:

```
PCC.EXE -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfile.c
```

- You can place more than one of each option on the command line, like this:

```
PCC.EXE -Ldirname1 -Ldirname2 -Iinc1 -Iinc2 -Iinc3 myfile.c
```

- You can mix listings:

```
PCC.EXE -Ldirname1;dirname2 -Iinc1 -Ld:dirname3 -Iinc2;inc3  
myfile.c
```

If you list multiple **-L** or **-I** options on the command line, the result is cumulative; the compiler searches all the directories listed in order from left to right.

File search algorithms

#include-file search algorithms

Paradigm C++ searches for files included in your source code with the **#include** directive in the following ways:

If you specify a path and/or directory with your include statement, Paradigm C++ searches only the location specified. For example, if you have the following statement in your code:

```
#include "c:\PARADIGM\include\stdio.h"
```

the header file `stdio.h` must reside in the directory `C:\PARADIGM\INCLUDE`. In addition, if you use the statement:

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

and you set the Include option (**-I**) to specify the path `c:\PARADIGM\include`, the file `stdio.h` must reside in `C:\PARADIGM\INCLUDE`.

- If you put an `#include <somefile>` statement in your source code, Paradigm C++ searches for "somefile" only in the directories specified with the Include (**-I**) option.
- If you put an `#include "somefile"` statement in your code, Paradigm C++ first searches for "somefile" in the current directory; if it does not find the file there, it then searches in the directories specified with the Include (**-I**) option.

Library file search algorithms

The library file search algorithms are similar to those for include files:

- Implicit libraries: Paradigm C++ searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for `#include <somefile>`.

Implicit library files are the ones Paradigm C++ automatically links in and the start-up object file (C0x.OBJ). To see these files in the Project Manager, turn on run-time nodes (choose Options|Environment|Project View, then check Show Runtime Nodes).

- Explicit libraries: Where Paradigm C++ searches for explicit (user-specified) libraries depends in part on how you list the library file name. Explicit library files are ones you list on the command line or in a project file; these are file names with a .LIB extension.
- If you list an explicit library file name with no drive or directory (like this: mylib.lib), Paradigm C++ first searches for that library in the current directory. If the first search is unsuccessful, Paradigm C++ looks in the directories specified with the Library (-L) option. This is similar to the search algorithm for #include "somefile".
- If you list a user-specified library with drive and/or directory information (like this: c:\mystuff\mylib1.lib), Paradigm C++ searches only in the location you explicitly listed as part of the library path name and not in any specified library directories.

Output directories

The Output Directories options specify the directories where your .OBJ, .AXE, .EXE, and .MAP files are placed. The Paradigm C++ IDE looks for those directories when performing a make or run and to check dates and times of .OBJs, .AXEs, and .EXEs. If the entry is blank, the files are stored in the current directory.

Click the down-arrow icon or press *Alt+Down* arrow to display the history list of previously entered directory names.



You can use the **\$INHERIT** and **\$ENV()** macros in any of the following input fields.

Intermediate

Use the Intermediate list box to specify where Paradigm C++ places object (.OBJ) and map (.MAP) files when it builds your project. This is also the directory where a tool places any temporary files that it might create.

Final

Command-line equivalent: **-npath**, where *path* = directory path

The Final list box specifies the location where the Paradigm C++ IDE places the generated target files (for example, .AXE and .EXE files).

Guidelines for entering directory names

Use the following guidelines when entering directories in the Directories options pages.

- You must separate multiple directory path names (if allowed) with a semicolon (;).
- You can use up to a maximum of 127 characters (including whitespace).
- Whitespace before and after the semicolon is allowed but not required.
- Relative and absolute path names are allowed, including path names relative to the logged position in drives other than the current one.

For example,

```
C:\;C:\..\PARADIGM;D:\myprog\source
```

\$INHERIT and \$ENV()

Paradigm C++ supports the two macros `$INHERIT` and `$ENV()` in the Directories page, the Compiler|Defines page and the Compiler|Precompiled Header page of the Project Options dialog box.



You can add `$INHERIT` and `$ENV()` anywhere in the strings you type into the input boxes.

\$INHERIT

The `$INHERIT` macro expands to the value of the respective option of the current nodes parent.

For example, suppose the project node `MYSOURCE.CPP` has a parent node `MYSOURCE.AXE`, and the defines for `MYSOURCE.AXE` are

```
WIN31;
```

If you set the Defines value for `MYSOURCE.CPP` to:

```
_RTLDLL;$INHERIT;STRICT
```

`MYSOURCE.CPP` will inherit the defines of `MYSOURCE.AXE`, which will give it the following Defines values:

```
_RTLDLL;WIN31;;STRICT
```

\$ENV()

The `$ENV(environment_variable)` macro expands to the defined value of the specified environment variable. For example, suppose the environment variable `PCROOT` is set to the following value:

```
PCROOT = C:\PARADIGM
```

You can then set the Include path in the Directories page as follows:

```
$ENV(PCROOT)\Include
```

This will set the actual include path to:

```
C:\PARADIGM\Include
```

Librarian options

Librarian options affect the behavior of the built-in librarian. The built-in librarian combines the `.OBJ` files in your project into `.LIB` files. Options in this section control that process. In addition, you can cause the librarian to generate a list (`.LST`) file containing the `.OBJS` in a generated `.LIB` and the functions those `.OBJS` contain.

`PLIB.EXE` is the command-line librarian.

Case-sensitive library

Command-line equivalent = `/C`

When the Case-sensitive library option is on, the librarian treats case as significant in all symbols in the library. For example, if Case-sensitive library is checked, `"CASE"`, `"Case"`, and `"case"` are all treated as different symbols.

Create extended dictionary

Command-line equivalent = **/E**

When the Create extended dictionary option is on, the librarian includes, in compact form, additional information that helps the linker process library files faster.

Generate list file

When the Generate list file option is on, the librarian automatically produces a list file (.LST) that lists the contents of your library when it is created.

Library page size

Command-line equivalent = **/Psize**, where *size* is number of pages

The Library page size input box is where you set the number of bytes in each library "page" (dictionary entry).

The page size determines the maximum size of the library. Page size must be a power of 2 between 16 and 32,768 inclusive. The default page size of 16 allows a library of about 1 MB in size.

To create a larger library, change the page size to the next higher value (32).

Purge comment records

Command-line equivalent = **/O**

When the Purge comment records option is on, the librarian removes all comment records from modules added to the library.

Linker options

Linker options affect how an application is linked.

Linker options let you control how intermediate files (.OBJ, and .LIB) are combined into absolute executables (.AXE) and dynamic-link libraries (.DLL). For most options in this section, you will usually want to keep the default settings.

16-bit linker

16-bit Linker options tell the linker how to link 16-bit programs.

Discard nonresident name table

Command-line equivalent = **/Gn**, 16-bit only

When the Discard Nonresident Name Table option is enabled, the linker does not emit the nonresident name table. The resultant image will contain only the module description in the nonresident names table.



See “*Transfer resident names to nonresident names table*,” page 3-87 for usage details.

Default = OFF

Enable 32-bit processing

Command-line equivalent = **/3**, 16-bit only

The Enable 32-bit processing option lets you link 32-bit DOS object modules produced by PASM or a compatible assembler. This option increases the memory requirements for PLINK and slows down linking.

Default = OFF

Inhibit optimizing far call to near

Command-line equivalent = **/f**, 16-bit only

When the linker patches two code segments together, and far calls are made from one to the other, the linker will optimize the code by converting the far calls to near calls. When Inhibit optimizing far call to near is enabled, this optimization does not occur.

You might want to enable this option when you experience run-time crashes that appear to be related to corrupt virtual tables. Because virtual tables reside in the code segment, their contents can sometimes be interpreted by the linker as one of these far calls.

Default = OFF

Initialize segments

Command-line equivalent = **/i**, 16-bit only

When the Initialize segments option is on, the linker initializes uninitialized trailing segments to be output into the executable file even if the segments do not contain data records. This is normally not needed and will increase the size of your .AXE files.

Default = OFF

Segment alignment

Command-line equivalent = **/A:dd**, 16-bit only

Use the Segment alignment input box to change the current byte value on which to align segments. The operating system seeks pages for loading based on this alignment value. You can enter numbers in the range of 2 to 65,535.



The alignment factor is automatically rounded up to the nearest power of two. For example, if you enter 650, it is rounded up to 1,024 (this is different from the 32-bit Segment Alignment option).

For efficiency, you should use the smallest value that still allows for correct segment offsets in the segment table.

Default = 512

Transfer resident names to nonresident names table

Command-line equivalent = **/Gr**, 16-bit only

This option causes the linker to copy all names in the resident names table which have not been specified as RESIDENTNAME in the .DEF file to the nonresident names table. The resultant image contains only the module name and the symbol names of the exported symbols that were specified as RESIDENTNAME in the .DEF file.

When you use this option, you *must* also specify the WEP entry point as a RESIDENTNAME in the EXPORTS section of the .DEF file (Windows obtains the WEP entry point for this symbol by looking it up in the resident names table).



When building .DLLs that contain many exports, it's possible to exceed the 64K header file limitation. Because the .DLL contains the resident names table in its header, moving

the exports out of the header using the **/Gr** option usually remedies this problem. The **/Gr** option causes the linker to transfer the names in the resident names table to the nonresident names table. Names in the nonresident names table are then assigned ordinal numbers, which your .EXE file uses when referencing the entry points in the .DLL.

There are two ways to create input files for the linker:

- Run IMPLIB on the .DLL to create an import library for linking purposes.
- Run IMPDEF in the .DLL to create a .DEF file for linking purposes.

Once the import library or .DEF file has been created, there is no need to keep the names in either the resident or the nonresident names tables. Relinking the .DLL and specifying *both* the Transfer resident names to nonresident names table (**/Gr**) and Discard nonresident name table (**/Gn**) options causes the linker to build a .DLL with an “empty” names table. Not only does this post-processing avoid the problem of exceeding the header limitation, but it also creates a .DLL that loads faster (because it’s smaller) and runs faster (because references to entry points are by ordinal number instead of by name).

To summarize this process, you must

1. Enable the **/Gr** switch to transfer the names in the resident names table to the nonresident names table. This also assigns ordinal numbers to the names. However; before doing so, make sure you have included a .DEF file with the following export definition in the EXPORTS section:

```
EXPORTS
    WEP @1 RESIDENTNAME
```

2. Build the .DLL.
3. Run IMPLIB or IMPDEF on the new .DLL file.
4. Enable the **/Gn** switch (along with the already enabled **/Gr** switch).
5. Relink the .DLL.

To see an example of this process, refer to the makefile that builds the ObjectWindows example programs.

Default = OFF

16-bit optimizations

The 16-bit optimizations control how the linker optimizes 16-bit .EXE programs. In most cases the final executable file size is reduced, which results in a faster load time.

Whenever you use one or more of these options, the linker reorders the .EXE segments as follows:

- PRELOAD segments
- PRELOAD resources
- LOAD ON CALL segments
- LOAD ON CALL resources



These options work only with 16-bit Windows and DPMI programs.

Chain fixup

Command-line equivalent = **/Oc**, 16-bit only

Chain fixups remove duplicate and/or unnecessary fixup data from the .EXE file. This is done by emitting only one fixup record for each unique internal fixup and "remembering" the duplicate fixups by creating a linked list of the internal fixup locations within the .EXE data segment. When the loader loads the .EXE, it applies the fixup specified in the fixup record to each of the locations specified in the linked list. Specifying this optimization also causes trailing zeros in data segments to be eliminated. This usually results in a significantly smaller .EXE file, which loads faster.

Default = OFF

Iterate data

Command-line equivalent = **/Oi**, 16-bit only

This option scans data segments for patterns of data (for example, a block with 128 bytes filled with "0"). Instead of emitting the data, PLINK emits a "description" of the block of data which matches the pattern (for example, a 5-byte descriptor specifying a 128 bytes of 0). Specifying this optimization also causes trailing zeros in data segments to be eliminated. This usually results in a significantly smaller .EXE file, while loads faster.

Default = OFF

Minimize resource alignment

Command-line equivalent = **/Or**, 16-bit only

This optimization switch is the same as the Minimize segment alignment switch (**/Oa**), except that it applies to resource alignment values instead of segment alignment values.

Default = OFF

Minimize segment alignment

Command-line equivalent = **/Oa**, 16-bit only

This optimization switch determines the minimum segment alignment value by examining the size of the .EXE file. An .EXE that has a size of 1 byte to 64K bytes results in an alignment value of 1; if the .EXE file size is 64K+1 bytes to 128K bytes, the alignment value is 2; and so on.

While this optimization results in a smaller .EXE file, the .EXE might load slower because the newly calculated alignment value may cause the segments to cross physical disk sector boundaries more often. Unless you have also specified the Segment Alignment (**/A**) linker option, the linker initially generates an .EXE using the default alignment value of 512. Note that this option overrides whatever alignment value the linker might have used to initially generate the .EXE file.

Default = OFF

32-bit linker

32-bit linker options tell the linker how to link 32-bit programs.

Allow import by ordinal

(Command-line equivalent = **/o**, 32-bit only)

This option is different than the 16-bit /O (overlays) option.

This option lets you import by ordinal value instead of by the import name. When you specify this option, the linker emits only the ordinal numbers (and not the import names) to the resident or nonresident name table for those imports that have an ordinal number specified. If you do not specify this option, the linker ignores all ordinal numbers contained in import libraries or the .DEF file, and emits the import names to the resident and nonresident tables.

Committed stack size (in hexadecimal)

Command-line equivalent = **/Sc:xxxx**, 32-bit only

Specifies the size of the committed stack in hexadecimal. The minimum allowable value for this field is 4K (0x1000) and any value specified must be equal to or less than the Reserved StackSize setting (**/S**).

Specifying the committed stack size here overrides any STACKSIZE setting in a module definition file.(add note sidebar)

The command-line version of this option (**/Sc:xxxx**) accepts hexadecimal numbers as the stack reserve value.

Default = 8K (0x2000)

Committed heap size (in hexadecimal)

Command-line equivalent = **/Hc:xxxx**, 32-bit only

Specifies the size of the committed heap in hexadecimal. The minimum allowable value for this field is 0 and any value specified must be equal to or less than the Reserved Heap Size setting (**/H**).

Specifying the committed heap size here overrides any HEAPSIZ setting in a module definition file.(add note sidebar)

The command-line version of this option (**/Hc:xxxx**) accepts hexadecimal numbers as the stack reserve value.

Default = 4K (0x1000)

File alignment (in hexadecimal)

Command-line equivalent = **/Af:xxxx**, 32-bit only

The File Alignment option specifies page alignment for code and data within the executable file. The linker uses the file alignment value when it writes the various objects and sections (such as code and data) to the file. For example, if you use the default value of 0x200, the linker stores the section of the image on 512-byte boundaries within the executable file.

When using this option, you must specify a file alignment value that is a power of 2, with the smallest value being 16.

The old style of this option (**/A:dd**) is still supported for backward compatibility. With this option, the decimal number **dd** is multiplied by the power of 2 to calculate the file alignment value.(add note sidebar)

The command-line version of this option (**/Af:xxxx**) accepts either decimal or hexadecimal numbers as the file alignment value.

Default = 512 (0x200)

Image base address (in hexadecimal)

Command-line equivalent = **/B:xxxx**, 32-bit only

The Image Base Address option specifies an image base address for an application, and is used in conjunction with the Image is based option. If this setting is turned on, internal fixes are removed from the image and the requested load address of the first object in the application is set to the hexadecimal number specified. All successive objects are aligned on 64K linear address boundaries. This option makes applications smaller on disk and improves both load-time and run-time performance (the operating system no longer has to apply internal fixes).

The command-line version of this option (**/B:xxxx**) accepts either decimal or hexadecimal numbers as the image base address.

It is not recommended that you enable this option when producing a DLL. In addition, do not use the default setting of 0x400000 if you intend to run your application of Win32s systems.(add note sidebar)

Default = 0x400000 (recommended for true Win32 system applications)

Image is based

The Image is Based option affects whether an application has an image base address. If this setting is turned on, internal fixes are removed from the image and the requested load address of the first object in the application is set to the number specified in the Image Base Address input box. Using this option can greatly reduce the size of your final application module; however, it is not recommended for use when producing a DLL.

Default = OFF

Maximum linker errors

Command-line equivalent = **/Enn**

Specifies maximum errors the linker reports before terminating. **/E0** (default) reports an infinite number of errors (that is, as many as possible).

Object alignment (in hexadecimal)

Command-line equivalent = **/Ao:xxxx**, 32-bit only

The linker uses the object alignment value to determine the virtual addresses of the various objects and sections (such as code and data) in your application. For example, if you specify an object alignment value of 8192, the linker aligns the virtual addresses of the sections in the image on 8192-byte (0x2000) boundaries.

When using this option, you must specify an object alignment value that is a power of 2, with the smallest value being 4096 (the default).

The command-line version of this option (**/Ao:xxxx**) accepts either decimal or hexadecimal numbers as the object alignment value.

Default = 4096 (0x1000)

Reserved heap size (in hexadecimal)

Command-line equivalent = **/H:xxxx**, 32-bit only

Specifies the size of the reserved heap in hexadecimal. The minimum allowable value for this field is 0.

Specifying the reserved heap size here overrides any HEAPSIZE setting in a module definition file.(add note sidebar)

The command-line version of this option (**/H:xxxx**) accepts hexadecimal numbers as the stack reserve value.

Default = 1Mb (0x1000000)

Reserved stack size (in hexadecimal)

Command-line equivalent = **/S:xxxx**, 32-bit only

Specifies the size of the reserved stack in hexadecimal. The minimum allowable value for this field is 4K (0x1000).

Specifying the reserved stack size here overrides any STACKSIZE setting in a module definition file.(add note sidebar)

The command-line version of this option (**/S:xxxx**) accepts hexadecimal numbers as the stack reserve value.

Default = 1Mb (0x1000000)

Use incremental linker

Uses the incremental linker.

When on, the first link of the executable file takes about the same amount of time as without the incremental linker. On subsequent links, when you make small changes to your source code, the link increases in speed. With the incremental linker in use, the link is usually less than 2 seconds, even for multiple megabyte images.

Verbose

Command-line equivalent = **/r**, 32-bit only

This option causes the linker to emit messages that indicate what part of the link cycle is currently being executed by the linker. With this option turned on, the linker emits some or all of the following messages:

- Starting pass 1
- Generating map file
- Starting pass 2

General

Use the Linker|General options to include or exclude debugging information from your .AXE or .DLL. Debug information must be included in your program if you want to use the debugger (you can turn it off for production versions).

Case-sensitive exports and imports

Command-line equivalent = **/C**, 16-bit only

When the Case-Sensitive Exports option is on, the linker is case sensitive when it processes the names in the IMPORTS and EXPORTS sections of the *module definition file*.

Use this option when you are trying to export non-callback functions from DLLs, as in exported C++ member functions or dynamic versions of ObjectWindows Library and BIDS.

Do not use this option for normal Windows callback functions (declared FAR PASCAL).

Default = OFF

Case-sensitive link

Command-line equivalent = `/c`

When the Case-Sensitive Link option is enabled, the linker differentiates between upper and lower-case characters in public and external symbols. Normally, this option should be checked, since C and C++ are both case-sensitive languages.

Default = ON

Code pack size

Command-line equivalent = `/P=n`, 16-bit only

Use Code Pack Size to change the default code-packing size to any value between 1 and 65,536. (On the command line, set *n* to a value between 1 and 65,536.)

You would probably want the limit to be a multiple of 4K under the 386 enhanced mode because of the paging granularity of the system. Although the optimum segment size in 386 enhanced mode is 4K, the default code segment packing size is 8K because typical code segments are from 4K to 8K in size, and the default of 8K might pack more efficiently.

Code segment packing typically increases performance because each maintained segment requires system overhead. On the command-line, `/P-` turns code segment packing off, which can be useful if you've turned it on in the configuration file, but want to turn it off for a particular link.

Default = 8192 bytes (8K)

Default libraries

Command-line equivalent = `/n`

When you are linking with modules created by a compiler other than the Paradigm C++ compiler, the other compiler might have placed a list of default libraries in the object file.

When the Default Libraries option is unchecked (off), the linker tries to find any undefined routines in these libraries and in the default libraries supplied by the Paradigm C++ IDE.

When this option is checked (on), the linker searches only the default libraries supplied by the Paradigm C++ IDE and ignores any defaults in .OBJ files. You might want to check this option when linking modules written in another language.

Default = ON

Include debug information

Command-line equivalent = `/v`

When the Include Debug Information option is on, the linker includes information in the output file needed to debug your application with Paradigm DEBUG.

On the command line, this option causes the linker to include debugging information in the executable file for all object modules that contain debugging information. You can use the `/v+` and `/v-` options to selectively enable or disable debugging information on a module-by-module basis (but not on the same command-line where you use `/v`). For example, the

following command includes debugging information for modules mod2 and mod3, but not for mod1 and mod4:

```
PLINK mod1 /v+ mod2 mod3 /v- mod4
```

Default = ON in the Paradigm C++ IDE; OFF on the command line

Pack code segments

Command-line equivalent = **/P**

Pack Code Segments has different meanings for 16-bit and 32-bit applications. In addition, Code Segment Packing applies only to 32-bit applications and DLLs.

For 16-bit links, Code Segment Packing causes the linker to minimize the number of code segments by packing as many code segments as possible into one physical segment up to (and never greater than) the code-segment packing limit, which is set to 8,192 (8K) by default. PLINK starts a new segment if needed.

Because there is a certain amount of system overhead for every segment maintained, code segment packing typically increases performance by reducing the number of segments.

For 32-bit links, Code Packing Segments means the linker packs all code into one “segment.” On the command-line, **/P-** turns this option off.

Default = ON

Subsystem version (major.minor)

Command-line equivalent = **/Vd.d**

This option lets you specify the Windows version ID on which you expect your application will be run. The linker sets the Subsystem version field in the .EXE header to the number you specify in the input box.

You can also set the Windows version ID in the SUBSYSTEM portion of the *module definition file* (.DEF file) However, any version setting you specify in the Paradigm C++ IDE or on the command line overrides the setting in the .DEF file.

Command-line usage

When you use the **/Vd.d** command-line option, the linker sets the Windows version ID to the number specified by **d.d..** For example, if you specify **/V4.0**, the linker sets the Subsystem version field in the .EXE header to 4.0, which indicates a Windows 95 application.

Default = 3.1

Map file

Linker|Map File options tell the linker what type of map file to produce. You specify the type of map file created with the Map File options. These options control the information generated on segment ordering, segment sizes, and public symbols.

Include source line numbers

Command-line equivalent: **/I**, 16-bit only

When the Include Source Line Numbers option is on, the linker includes source line numbers in the object map files.

For this option to work, linked .OBJ files must be compiled with debug information using **-v**.

When Include Source Line Numbers is on, make sure you turn *Jump Optimizations* off in the Optimization|16 bit Specific options page, otherwise the compiler might group together common code from multiple lines of source text during jump optimization, or it might reorder lines (which makes line-number tracking difficult).

Default = OFF

Map file

You use the Map File options to choose the type of map file to be produced at link time.

For settings other than Off, the map file is placed in the output directory defined in the Directories|Output page.

Off Command-line equivalent = /x

The Off option tells the linker not to create a map file.

Default = OFF

Segments Command-line equivalent = /s

The Segments option adds a “Detailed map of segments” to the map file created with the Publics option (/m). The detailed list of segments contains the segment class, the segment name, the segment group, the segment module, and the segment ACBP information. If the same segment appears in more than one module, each module appears as a separate line.

The ACBP field encodes the A (alignment), C (combination), and B (big) attributes into a set of four bit fields, as defined by Intel. PLINK uses only three of the fields: A, C, and B. The ACBP value in the map is printed in hexadecimal. The following field values must be ORed together to arrive at the ACBP value printed.

Table 3-1
Segment field
values

Field	Value	Description
A (alignment)	00	An absolute segment
	20	A byte-aligned segment
	40	A word-aligned segment
	60	A paragraph-aligned segment
	80	A page-aligned segment
	A0	An unnamed absolute portion of storage
C (combination)	00	Cannot be combined
	08	A public combining segment
B (big)	00	Segment less than 64K
	02	Segment exactly 64K

With the Segments options enabled, public symbols with no references are flagged idle. An idle symbol is a publicly defined symbol in a module that was not referenced by an EXTDEF record or by any other module included in the link. For example, this fragment from the public symbol section of a map file indicates that symbols Symbol11 and Symbol13 are not referenced by the image being linked (they can either be deleted or declared static since no other module requires these symbols):

0002:00000874	Idle	Symbol1
0002:00000CE4		Symbol2
0002:000000E7	Idle	Symbol3

Publics **Command-line equivalent = /m**

This option causes the linker to produce a map file that contains an overview of the application segments and two listings of the public symbols. The segments listing has a line for each segment, showing the segment starting address, segment length, segment name, and the segment class. The public symbols are broken down into two lists, the first showing the symbols in sorted alphabetically, the second showing the symbols in increasing address order. Symbols with absolute addresses are tagged Abs.

A list of public symbols is useful when debugging: many debuggers use public symbols, which lets you refer to symbolic addresses while debugging.



For more information, see Linker|Map file.

Print mangled names in map file

Command-line equivalent = /M

Prints the mangled C++ identifiers in the map file, not the full name. This can help you identify how names are mangled (mangled names are needed as input by some utilities).

Default = OFF

Warnings

Warnings options enable or disable the display of Linker warnings.

32-bit warnings

- No entry point
- Duplicate symbol
- No def file
- Import does not match previous definition
- Extern not qualified with `_import`
- Using based linking in DLL
- Self-relative fixup overflowed
- .EXE module built with a .DLL extension
- Multiple stack segments found

"No stack" warning

Use the PLINK32 command-line option /w-stk to turn this warning off.

This option lets you control whether or not the linker emits the "No stack" warning. The warning is generated if no stack segment is defined in any of the object files or in any of the libraries included in the link. Except for .DLLs, this indicates an error. If a Paradigm C++ program produces this error, make sure you are using the correct C0x startup object file.

Default = OFF

Warn duplicate symbol in .LIB

Command-line equivalent = /d 16-bit, /wdpl 32-bit

Use the *PLINK32* command-line option */w-dpl* to turn this warning off.

When the Warn Duplicate Symbols option is on, the linker warns you if a symbol appears in more than one object or library files.

If the symbol must be included in the program, the linker uses the symbol definition from the first file it encounters with the symbol definition.

Default = OFF

Make options

Make options control the conditions under which the building of a project stops and how the Project Manager uses autodependency information.

Autodependencies

When the Make|Autodependencies option is selected, the Project Manager automatically checks dependencies for every target that has a corresponding source file in the project list.

None

When None is selected, no autodependency checking is performed.

Use

When Use is selected, autodependency checking is performed by reading the autodependency information out of the .OBJ files.

Cache

When Cache is selected, autodependency information is stored in memory to make dependency checking faster. This option speeds up compilation, but autodependency information will not display in the Project Tree.

Cache and display

When Cache and Display is selected, the Project Manager stores the autodependency information in the project file. Once the autodependency information is generated (after a compile) the information is displayed in the Project Tree. This makes dependency checking faster, but makes project files larger.

Break make on

The Make|Break Make On options specify the error condition that stops the making of a project.

Warnings

This option stops a make if the compiler encounters warnings.

Command-line equivalent = **-w!**

When this compiler option is enabled, the compiler terminates the compile and returns a non-zero error code if a warning is encountered; an .OBJ file is not created.

Errors

This option stops a make when the compiler encounters errors.

Fatal errors

This option tells the Project Manager to generate a list of errors and warnings for all files and all targets in the project. The Project Manager will go on to link if no errors occur.

Default = Errors

New node path

Turn on the Absolute option if you want new nodes to have an absolute, instead of a relative, path.

Messages options

Messages options let you control the messages generated by the compiler. Compiler messages are indicators of potential trouble spots in your program. These messages can warn you of many problems that may be waiting to happen, such as variables and parameters that are declared but never used, type mismatches, and many others.

Setting a message option causes the compiler to generate the associated message or warning when the specific condition arises. Note that some of the messages are on by default.

ANSI violations

Compiler Messages|ANSI Violations options enable or disable individual warning messages about statements that violate the ANSI standard for the C language.

Table 3-2
*ANSI violation
messages*

Option	Command-line equivalent	Default
Void functions may not return a value	-w-voi	ON
Both return and return of a value used	-w-ret	ON
Suspicious pointer conversion	-w-sus	ON
Undefined structure 'ident'	-wstu	OFF
Redefinition of 'ident' is not identical	-w-dup	ON
Hexadecimal value more than three digits	-w-big	ON
Bit fields must be signed or unsigned int	-wbbf	OFF
'ident' declared as both external and static	-w-ext	ON
Declare 'ident' prior to use in prototype	-w-dpu	ON
Division by zero	-w-zdi	ON
Initializing 'ident' with 'ident'	-w-bei	ON
Initialization is only partially bracketed	-w-pin	OFF
Non-ANSI keyword used	-wnak	OFF

Display warnings

Use the Display Warnings options to choose which warnings are displayed.

All

Command-line equivalent: **-w**

Display all warning and error messages.

Default = OFF

Selected

Command-line equivalent: **-waaa**

Choose which warnings are displayed. Using `pragma warn` in your source code overrides messages options set either at the command line or in the Paradigm C++ IDE.

To disable a message from the command line, use the command-line option **-w-aaa**, where **aaa** is the 3-letter message identifier used by the command-line option.

Default = ON

None

Suppresses the display of warning messages. Errors are still displayed.

Default = OFF

General

Compiler Messages|General options enable or disable a few general warning messages.

Table 3-3
General warning
messages

Option	Command-line equivalent	Default
Unknown assembler instruction	-wasm	OFF
Ill-formed pragma	-w-ill	ON
Array variable 'ident' is near	-w-ias	ON
Superfluous & with function	-wamp	OFF
'ident' is obsolete	-w-obs	ON
Cannot create precompiled header	-w-pch	ON
User-defined warnings	-w-msg	ON

User-defined warnings

Command-line equivalent: **-wmsg**

The User-defined warnings option allows user-defined messages to appear in the Paradigm C++ IDE's Message window. User-defined messages are introduced with the `#pragma message` compiler syntax.



In addition to messages that you introduce with the `#pragma message` compiler syntax, User-defined warnings displays warnings introduced by third-party libraries. Remember, if you need Help on a third-party warning, please contact the vendor of the header file that issued the warning.

Default = ON

Inefficient C++ coding

Compiler Messages|Inefficient C++ Coding options enable or disable individual warning messages about inefficient C++ coding.

Table 3-4
Inefficient C++
coding
messages

Option	Command-line equivalent	Default
Functions containing 'ident' not expanded inline	-w-inl	ON
Temporary used to initialize 'ident'	-w-lin	ON
Temporary used for parameter 'ident'	-w-lvc	ON

Inefficient coding

Compiler Messages|Inefficient Coding options are used to enable or disable individual warning messages about inefficient coding.

Table 3-5
Inefficient coding
messages

Option	Command-line equivalent	Default
'ident' assigned a value which is never used	-w-aus	ON
Parameter 'ident' is never used	-w-par	ON
'ident' declared but never used	-w-use	OFF
Structure passed by value	-w-stv	OFF
Unreachable code	-w-rch	ON
Code has no effect	-w-eff	ON



The warnings `Unreachable Code` and `Code Has No Effect` can indicate serious coding problems. If the compiler generates these warnings, be sure to examine the lines of code that cause these warnings.

Obsolete C++

Compiler Messages|Obsolete C++ options choose which specific obsolete items or incorrect syntax C++ warnings to display.

Table 3-6
Obsolete C++
messages

Option	Command-line equivalent	Default
Base initialization without class name is obsolete	-w-obi	ON
This style of function definition is obsolete	-w-ofp	ON
Overloaded prefix operator used as postfix operator	-w-pre	ON

Portability

Compiler Messages|Portability options enable or disable individual warning messages about statements that might not operate correctly in all computer environments.

Table 3-7
Portability
messages

Option	Command-line equivalent	Default
Non-portable pointer conversion	-w-rpt	ON
Non-portable pointer comparison	-w-cpt	ON
Constant out of range in comparison	-w-rng	ON
Constant is long	-w-cln	OFF
Conversion may lose significant digits	-wsig	OFF
Mixing pointers to signed and unsigned char	-wucp	OFF

Potential C++ errors

Compiler Messages|Potential C++ Errors options enable or disable individual warning messages about statements that violate C++ language implementation.

Table 3-8
Potential
C++Errors

Option	Command-line equivalent	Default
Constant member 'ident' is not initialized	-w-nci	ON
Assigning 'type' to 'enumeration'	-w-eas	ON
'function' hides virtual function 'function2'	-w-hid	ON

Non-const function <function> called for const object	-w-ncf	ON
Base class 'ident' inaccessible because also in 'ident'	-w-ibc	ON
Array size for 'delete' ignored	-w-dsz	ON
Use qualified name to access nested type 'ident'	-w-nst	ON
Handler for '<type1>' Hidden by Previous Handler for '<type2>'	-w-hch	ON
Conversion to 'type' will fail for virtual base members	-w-mpc	ON
Maximum precision used for member pointer type <type>	-w-mpd	ON
Use '> >' for nested templates instead of '>>'	-w-ntd	ON
Non-volatile function <function> called for volatile object	-w-nvf	ON

Potential errors

Compiler Messages|Potential Errors options enable or disable individual warning messages about potential coding errors.

Table 3-9
Potential error
messages

Option	Command-line equivalent	Default
Possibly incorrect assignment	-w-pia	ON
Possible use of 'ident' before definition	-wdef	OFF
No declaration for function 'ident'	-wnod	OFF
Call to function with no prototype	-w-pro	ON
Function should return a value	-w-rvl	ON
Ambiguous operators need parentheses	-wamb	OFF
Condition is always (true/false)	-w-ccc	ON
Continuation character \ found in //	-w-com	ON

Stop after ... errors

Entering 0
causes
compilation to
continue until the
end of the file.

Command-line equivalent: **-jn**

Errors: Stop After causes compilation to stop after the specified number of errors has been detected. You can enter any number from 0 to 255.

Default = 25

Stop after ... warnings

Command-line equivalent: **-gn**

Warnings: Stop After causes compilation to stop after the specified number of warnings has been detected. You can enter any number from 0 to 255.

Entering 0 causes compilation to continue until either the end of the file or the error limit set in Errors: Stop After has been reached, whichever comes first.

Default = 100

Optimization options

Optimization options are the software equivalent of performance tuning. There are two general types of compiler optimizations:

- Those that make your code smaller

- Those that make your code faster

Although you can compile with optimizations at any point in your product development cycle, be aware when debugging that some assembly instructions might be "optimized away" by certain compiler optimizations.

General settings

The main Optimizations page in the Project Options dialog box contains four radio buttons that let you select the overall type of optimizations you want to use. Because of the complexities of setting compiler optimizations, it is recommended that you use either the Optimize for Size or the Optimize for Speed radio buttons. The general optimization settings are:

- Disable all optimizations
- Use selected optimizations
- Optimize for size
- Optimize for speed

16- and 32-bit

The 16- and 32-bit compiler options specify optimization settings for all compilations.

Common subexpression

The Common subexpressions options tell the compiler how to find and eliminate duplicate expressions in your code.

No optimization	When the No optimization option is on, the compiler does not eliminate common subexpressions. This is the default behavior of the command-line compilers.
Optimize locally	<p>Command-line equivalent: -Oc</p> <p>When the Optimize locally option is on, the compiler eliminates common subexpressions within groups of statements unbroken by jumps (basic blocks).</p>
Optimize globally	<p>Command-line equivalent: -Og</p> <p>When you set this option, the compiler eliminates common subexpressions within an entire function. This option globally eliminates duplicate expressions within the target scope and stores the calculated value of those expressions once (instead of recalculating the expression).</p> <p>Although this optimization could theoretically reduce code size, it optimizes for speed and rarely results in size reductions. Use this option if you prefer to reuse expressions rather than create explicit stack locations for them.</p>

Induction variables

Command-line equivalent: **-Ov**

When this option is enabled, the compiler creates induction variables and it performs strength reduction, which optimizes for loops speed.

Use this option when you're compiling for speed and your code contains loops. The optimizer uses induction to create new variables (induction variables) from expressions used in loops. The optimizer assures that the operations performed on these new variables are computationally less expensive (reduced in strength) than those used by the original variables.

Optimizations are common if you use array indexing inside loops, because a multiplication operation is required to calculate the position in the array that is indicated by the index. For example, the optimizer creates an induction variable out of the operation `v[i]` in the following code because the `v[i]` operation requires multiplication. This optimization also eliminates the need to preserve the value of `i`:

```
int v[10];
void f(int x, int y, int z)
{
    int i;
    for (i = 0; i < 10; i++)
        v[i] = x * y * z;
}
```

With Induction variables enabled, the code changes:

```
int v[10];
void f(int x, int y, int z)
{
    int i, *p;
    for (p = v; p < &v[9]; p++)
        *p = x * y * z;
}
```

Inline intrinsic functions

Command-line equivalent: **-Oi**

When the Inline Intrinsic Functions option is on, the compiler generates the code for common memory functions like **strcpy()** within your function's scope. This eliminates the need for a function call. The resulting code executes faster, but it is larger.

The following functions are inlined with this option:

alloca	fabs	memchr	memcmp
memcpy	memset	rotl	rotr
strcpy	strcat	strchr	strcmp
strcpy	strlen	strncat	strncmp
strncpy	strnset	strchr	

You can control the inlining of these functions with the pragma **intrinsic**. For example, `#pragma intrinsic strcpy` causes the compiler to generate inline code for all subsequent calls to `strcpy` in your function, and `#pragma intrinsic -strcpy` prevents the compiler from inlining `strcpy`. Using these pragmas in a file overrides any compiler option settings.

When inlining any intrinsic function, you must include a prototype for that function before you use it; the compiler creates a macro that renames the inlined function to a function that the compiler recognizes internally. In the previous example, the compiler would create a macro `#define strcpy _strcpy_`.

The compiler recognizes calls to functions with two leading and two trailing underscores and tries to match the prototype of that function against its own internally stored prototype. If you don't supply a prototype, or if the prototype you supply doesn't match the compiler's prototype, the compiler rejects the attempt to inline that function and generates an error.

16-bit

The Optimizations|16-bit options pertain to 16-bit applications only.

Assume no pointer aliasing

Command-line equivalent: **-Oa**

When the Assume no pointer aliasing option is on, the compiler assumes that pointer expressions are not aliased in common subexpression evaluation.

Assume no pointer aliasing affects the way the optimizer performs common subexpression elimination and copy propagation by letting the optimizer maintain copy propagation information across function calls and by letting the optimizer maintain common subexpression information across some stores. Without this option the optimizer must discard information about copies and subexpressions. Pointer aliasing might create bugs that are hard to spot, so it is only applied when you enable this option.

Assume no pointer aliasing controls how the optimizer treats expressions that contain pointers. When compiling with global or local common subexpressions and Assume no pointer aliasing is enabled, the optimizer recognizes `*p * x` as a common subexpression in function *func1*.

```
int g, y;
int func1(int *p)
{
    int x=5;
    y = *p * x;
    g = 3;
    return (*p * x);
}
void func2(void)
{
    g=2;
    func1(&g); // This is incorrect--the assignment g = 3
               // invalidates the expression *p * x
}
```

Copy propagation

Command-line equivalent: **-Op**

When this option is enabled; copies of constants, variables, and expressions are propagated whenever possible.

Copy propagation is primarily speed optimization, but it never increases the size of your code. Like loop-invariant code motion, copy propagation relies on the analysis performed during common subexpression elimination. Copy propagation means that the optimizer remembers the values assigned to expressions and uses those values instead of loading the value of the assigned expressions. With this, copies of constants, expressions, and variables can be propagated.

Dead code elimination

Command-line equivalent: **-Ob**

When the Dead code elimination option is on, the compiler reveals variables that might not be needed. Because the optimizer must determine where variables are no longer used (live range analysis), you might also want to set Global register allocation (**-Oe**) when you use this option.

Global register allocation

Command-line equivalent: **-Oe**

When this option is enabled, global register allocation and variable live range analysis are enabled. This option should always be used when optimizing code because it increases the speed and decreases the size of your application.

Invariant code motion

Command-line equivalent: **-Om**

When this option is enabled, invariant code is moved out of loops and your code is optimized for speed. The optimizer uses information about all the expressions in the function (gathered during common subexpression elimination) to find expressions whose values do not change inside a loop.

To prevent the calculation from being done many times inside the loop, the optimizer moves the code outside the loop so that it is calculated only once. The optimizer then reuses the calculated value inside the loop.

You should use loop-invariant code motion whenever you are compiling for speed and have used global common subexpressions, because moving code out of loops can result in enormous speed gains. For example, in the following code, $x * y * z$ is evaluated in every iteration of the loop:

```
int v[10];
void f(int x, int y, int z)
{
    int i;
    for (i = 0; i < 10; i++)
        v[i] = x * y * z;
}
```

The optimizer rewrites the code:

```
int v[10];
void f(int x, int y, int z)
{
    int i,t1;
    t1 = x * y * z;
    for (i = 0; i < 10; i++)
        v[i] = t1;
}
```

Jump optimization

Command-line equivalent: **-O**

When Jump optimization option is on, the compiler reduces the code size by eliminating redundant jumps and reorganizing loops and switch statements.

When this option is enabled, the sequences of *stepping* in the debugger can be confusing because of the reordering and elimination of instructions. If you are debugging at the assembly level, you might want to disable this option.

Default = ON

Loop optimization

Command-line equivalent: **-OI**

When this option is enabled, loops are compacted into REP/STOSx instructions.

Loop optimization takes advantage of the string move instructions on the 80x86 processors by replacing the code for a loop with a string move instruction, making the code faster.

Depending on the complexity of the operands, the compacted loop code can also be smaller than the corresponding non-compacted loop.

Suppress redundant loads

Command-line equivalent: **-Z**

When this option is enabled, the compiler suppresses the reloading of registers by remembering the contents of registers and reusing them as often as possible.

Exercise caution when using this option; the compiler cannot detect if a value has been modified indirectly by a pointer.

32-bit

Use the Optimizations|32-bit options to specify options specific to the Pentium processor and the Intel optimizing compiler. The options are:

Pentium instruction scheduling

Command-line equivalent: **-OS**

When enabled, this switch rearranges instructions to minimize delays that can be caused by Address Generation Interlocks (AGI) which occur on the i486 and Pentium processors. This option also optimizes the code so that it takes advantage of the Pentium parallel pipelines. Best results for Pentium systems are obtained when you use this switch in conjunction with the 32-bit Compiler|Pentium option in the Project Options dialog box (-5).



Scheduled code is more difficult to debug at the source level because instructions from a particular source line may be mixed with instructions from other source lines. Stepping through the source code is still possible, although the execution point might make unexpected jumps between source lines as you step. Also, setting a breakpoint on a source line may result in several breakpoints being set in the code. This is especially important to note when inspecting variables, since a variable may be undefined even though the execution point is positioned after the variable assignment.

Stepping through the following function when this switch is enabled demonstrates the stepping behavior:

```

int v[10];
void f(int i, int j)
{
    int a,b;

    a = v[i+j];
    b = v[i-j];
    v[i] = a + b;
    v[j] = a - b;
}

```

Execution starts by computing the index $i - j$ in the assignment to b (note that a is still undefined although the execution point is positioned after the assignment to a). The index $i + j$ is computed, $v[i - j]$ is assigned to b , and $v[i + j]$ is assigned to a . If a breakpoint is set on the assignment to b , execution will stop twice: once when computing the index and again when performing the assignment.

Default = OFF (**-O-S**)

Cache hit optimizations (Intel compiler only)

Command-line equivalent: **-OM**

Specifies a set of memory accessing optimizations that improves cache hits and reduces the number of memory accesses. These optimizations include

- Loop interchange
- Loop distribution
- Strip mining and preloading
- Loop blocking
- Alternate loops
- Loop unrolling

Optimize across function boundaries (Intel compiler only)

Command-line equivalent: **-OI**

Specifies a set of interprocedural optimizations. These optimizations eliminate call overhead and can create opportunities for further optimizations. They are applied across procedure boundaries but are restricted to routines within the same file, including routines in files combined by the **#include** preprocessor directive. These optimizations include

- Monitoring module-level static variables
- Inline function expansion
- Cloning
- Passing arguments in registers
- Constant argument propagation



Currently, these optimizations are disabled if your source code contains embedded assembly code.

General optimization settings

Disable all optimizations

Command-line equivalent: **-Od**

Disables all optimization settings, including ones which you may have specifically set and those which would normally be performed as part of the speed/size tradeoff.

Because this disables code compaction (tail merging) and cross-jump optimizations, using this option can keep the debugger from jumping around or returning from a function without warning, which makes stepping through code easier to follow.



You can override this setting using the predefined Style Sheets in the Project Manager.

Use selected optimizations

Does not set any optimization by default, but lets you set the specific optimization options you need through the settings contained in the Optimization subtopics. The subtopic pages are

- 16 and 32-bit
- 16-bit specific
- 32-bit specific



Configuring your own optimization settings should be reserved for expert users only.

Optimize for size

Command-line equivalents: **-O1**

This radio button sets an aggregate of optimization options that tells the compiler to optimize your code for size. For example, the compiler scans the generated code for duplicate sequences. When such sequences warrant, the optimizer replaces one sequence of code with a jump to the other and eliminates the first piece of code. This occurs most often with **switch** statements. The compiler optimizes for size by choosing the smallest code sequence possible.

This option (**-O1**) sets the following optimizations:

- Jump optimizations (**-O**)
- Dead code elimination (**-Ob**)
- Duplicate expressions (**-Oe**)
- Register allocation and live range analysis (**-Oe**)
- Loop optimizations (**-Ol**)
- Instruction scheduling (**-Os**)
- Register load suppression (**-Z**)



The compiler options **-Ot** and **-G** are supported for backward compatibility only, and are equivalent to the **-O1** compiler option.

Optimize for speed

Command-line equivalent: **-O2**

This radio button sets an aggregate of optimization options that tells the compiler to optimize your code for speed. This switch (**-O2**) sets the following optimizations:

- Dead code elimination (**-Ob**)

- Register allocation and live range analysis (**-Oe**)
- Duplicate expression within functions (**-Og**)
- Intrinsic functions (**-Oi**)
- Loop optimizations (**-Ol**)
- Code motion (**-Om**)
- Copy propagation (**-Op**)
- Instruction scheduling (**-OS**)
- Induction variables (**-Ov**)
- Register load suppression (**-Z**)



The compiler options **-Os** and **-G-** are supported for backward compatibility only, and are equivalent to the **-O2** compiler option. The **-Ox** option is also supported for backward compatibility and for compatibility with Microsoft make files.

Command-line only options

The options are available only from the command line.

Object search paths

Command-line equivalent = **/j**

This option lets you specify the directories the linker will search if there is no explicit path given for an .OBJ module in the compile/link statement. This option works with both PLINK and PLINK32.

The Specify object search path uses the following command-line syntax:

```
/j<PathSpec>[ ;<PathSpec>][...]
```

The linker uses the specified object search path(s) if there is no explicit path given for the .OBJ file and the linker cannot find the object file in the current directory. For example, the command

```
PLINK32 /jc:\myobjs;.\objs splash .\common\logo,,,utils logolib
```

directs the linker to first search the current directory for SPLASH.OBJ. If it is not found in the current directory, the linker then searches for the file in the C:\MYOBS directory, and then in the .\OBS directory. However, notice that the linker does not use the object search paths to find the file LOGO.OBJ because an explicit path was given for this file.

16- and 32-bit command-line options

The following command-line switches are supported by the command-line compiler PCC.EXE, PCC32.EXE, and PCC32i.EXE.

Compile to .ASM, then assemble

Command-line equivalent = **-B**

This command-line option causes the compiler to first generate an .ASM file from your C++ (or C) source code (same as the **-S** command-line option). The compiler then calls PASM (or the assembler specified with the **-E** option) to create an .OBJ file from the .ASM file. The .ASM file is then deleted. To use this 32-bit compiler option, you must install a 32-bit assembler, such as PASM32.EXE, and then specify this assembler with

the **-E** option. In the Paradigm C++ IDE, right-click the source node in the Project Manager, then choose Special|C++ to Assembler.



Your program will fail to compile with the **-B** option if your C or C++ source code declares static global variables that are keywords in assembly. This is because the compiler does not precede static global variables with an underscore (as it does other variables), and the assembly keywords will generate errors when the code is assembled.

Compile to .OBJ, no link

Command-line equivalent = **-c**

Compiles and assembles the named .C, .CPP, and .ASM files, but does not execute a link on the resulting .OBJ files. In the Paradigm C++ IDE, choose Project|Compile.

Specify assembler

Command-line equivalent = **-Efilename**

Assemble instructions using *filename* as the assembler. The 16-bit compiler uses PASM as the default assembler. In the Paradigm C++ IDE, you can configure a different assembler using the Tool menu.

Specify executable file name

Command-line equivalent = **-efilename**

Link file using *filename* as the name of the executable file. If you do not specify an executable name with this option, the linker creates an executable file based on the name of the first source file or object file listed in the command.

Pass option to linker

Command-line equivalent = **-lx**

Use this command-line option to pass option(s) *x* to the linker from a compile command. Use the command-line option **-l-x** to disable a specific linker option.

Create a MAP file

Command-line equivalent = **-M**

Use this command-line option tells the linker to create a map file.

Compile .OBJ to filename

Command-line equivalent = **-ofilename**

Use this option to compile the specified source file to *filename*.OBJ.

C++ compile

Command-line equivalent = **-P**

The **-P** command-line option causes the compiler to compile all source files as C++ files, regardless of their extension. Use **-P-** to compile all .CPP files as C++ source files and all other files as C source files.

The command-line option **-Pext** causes the compiler to compile all source files as C++ files and it changes the default extension to whatever you specify with *ext*. This option is provided because some programmers use different extensions as their default extension for C++ code.

The option **-P-*ext*** compiles files based on their extension (.CPP compiles to C++, all other extensions compile to C) and sets the default extension (other than .CPP).

Compile to assembler

Command-line equivalent = **-S**

This option causes the compiler to generate an .ASM file from your C++ (or C) source code. The generated .ASM file includes the original C or C++ source lines as comments in the file.

Specify assembler option

Command-line equivalent = **-Tx**

Use this command-line option to pass the option(s) *x* to the assembler you specify with the **-E** option. To disable all previously enabled assembler options, use the **-T-** command-line option.

Undefine symbol

Command-line equivalent = **-U*name***

This command-line option undefines the previous definition of the identifier *name*.

Linker supported command-line options

The following switches are supported by the 16-bit command-line compiler (PCC.EXE) and linker (PLINK.EXE).

Generate 8087 instructions

Command-line equivalent = **-f87**

Use this 16-bit compiler option to create 16-bit real-mode 8087 floating-point code.

Compile to 16-bit real-mode .AXE

Command-line equivalent = **-tD**

The compiler creates a 16-bit real-mode .AXE file (same as **-tDe**).

Enable backward compatibility options

Command-line equivalent = **-Vo**

This compiler option enables the following 16-bit backward compatibility options: **-Va**, **-Vb**, **-Vc**, **-Vp**, **-Vt**, **-Vv**. Use this option as a handy shortcut when linking libraries built with older versions of Paradigm C++.

Link 16-bit real-mode .EXE

Command-line equivalent = **/Tde**

PLINK generates a real-mode 16-bit real-mode .EXE file.

Extended memory swapping

Command-line equivalent = **/yx**

This PLINK option controls the linker's use of extended memory for I/O buffering. By default, the linker can use up to 8MB of extended memory. You can control the linker's use of extended memory with one of the following forms of this switch:

- **/yx** or **/yx+** uses all available extended memory, up to 8MB
- **/yxn** uses only up to *n* KB of extended memory

Default = OFF

Enable 24-bit extended addressing

Command-line equivalent = **-Y**

Enables use of the 24-bit extended addressing mode to allow a real-mode address space of 16MB.

When this option is enabled, the macro **__EXTADDR__** will be defined.

32-bit command-line options

The following switches are supported by the 32-bit command-line compilers (PCC32.EXE and PCC32i.EXE) and linker (PLINK32.EXE).



The following 32-bit command-line options are not needed if you include a module definition file in your compile and link commands which specifies the type of 32-bit application you intend to build.

Generate a multi-threaded target

Command-line equivalent = **-tWM**

The compiler creates a multi-threaded .EXE or .DLL. (The command-line option **-WM** is supported for backward compatibility only; it has the same functionality as **-tWM**.)

Link using 32-bit Windows API

Command-line equivalent = **/aa**

PLINK32 generates a protected-mode executable that runs using the 32-bit Windows API.

Link for 32-bit console application

Command-line equivalent = **/ap**

PLINK32 generates a protected-mode executable file that runs in console mode.

Link 32-bit .DLL file

Command-line equivalent = **/Tpd**

PLINK32 generates a 32-bit protected-mode Windows .DLL file.

Link 32-bit .EXE file

Command-line equivalent = **/Tpe**

PLINK32 generates a 32-bit protected-mode Windows .EXE file.

Compiler command-line options

The following table lists the command-line compiler options in alphabetical order:

Table 3-10
Compiler
command-line
options

Option	Description
@ filename	Read compiler options from the <i>response file</i> "filename"
+ filename	Use alternate <i>configuration file</i> "filename"
-1-	Generate 8086 compatible instructions (Default for 16-bit)
-1	Generate the 80186/286 compatible instructions (16-bit only)
-2	Generate 80286 protected-mode compatible instructions (16-bit compiler only)
-3	Generate 80386 protected-mode compatible instructions (Default for 32-bit)
-4	Generate 80386/80486 protected-mode compatible instructions
-5	Generate Pentium instructions
-A	Use only ANSI keywords
-a	Align byte (Default: -a- use byte-aligning)
-AK	Use only Kernighan and Ritchie keywords
-an	Align to "n" where 1=byte, 2=word (16-bit = 2 bytes) 4=Double word (32-bit only, 4 bytes), 8=Quad word (32-bit only, 8 bytes)
-AT	Use Paradigm C++ keywords (also -A-)
-AU	Use only UNIX V keywords
-B	Compile to .ASM (-S), the assemble to .OBJ (command-line compiler only)
-b	Make enums always integer-sized (Default: -b- make enums byte-sized when possible)
-C	Turn nested comments on (Default: -C- turn nested comments off)
-c	Compile to .OBJ, no link (command-line compiler only)
-D name	Define "name" to the null string
-D name = <i>string</i>	Define "name" to "string"
-d	Merge duplicate strings (Default)
-dc	Move string literals from data segment to code segment (16-bit compiler only)
-E filename	Specify assembler
-e filename	Specify executable file name
-f	Emulate floating point
-f-	No floating point
-f87	Generate 8087 floating-point code (command-line compiler only)
-Fa	Enable page alignment for far segments
-Fb	Enable Borland C++-compatible far data
-Fc	Generate COMDEFs (16-bit compiler only)
-Ff	Create far variables automatically
-Ff= <i>size</i>	Create far variables automatically; set the threshold to " <i>size</i> " (16-bit compiler only)
-ff	Fast floating point
-fp	Correct Pentium FDIV flaw
-Fs	Assume DS=SS in all memory models (16-bit compiler only)
-gn	Warnings: stop after "n" messages (Default: 255)
-H	Generate and use precompiled headers (Default)
-H= filename	Set the name of the file for precompiled headers

-H"xxx"	Stop precompiling after header file <i>xxx</i>
-h	Uses fast huge pointers
-Hc	Cache precompiled header
-Hu	Use but do not generate precompiled headers
-in	Make significant identifier length to be "n" (Default)
-Jg	Generate definitions for all template instances and merge duplicates (Default)
-Jgd	Generate public definitions for all template instances; duplicates result in redefinition errors
-Jgx	Generate external references for all template instances
-jn	Errors: stop after "n" messages (Default)
-K	Default character type unsigned (Default: -K- default character type signed)
-k	Turn on standard stack frame (Default)
-K2	Allow only two character types (signed and unsigned). Char is treated as signed.
-lx	Pass option "x" to linker (command-line compiler only)
-M	Create a Map file (command-line compiler only)
-mc	Compile using compact memory model (16-bit compiler only)
-mh	Compile using huge memory model
-ml	Compile using large memory model (16-bit compiler only)
-mm	Compile using medium memory model (16-bit compiler only)
-mm!	Compile using medium memory model; assume DS!=SS (16-bit compiler only. Note: there is no space between the -mm and the !)
-ms	Compile using small memory model (Default, 16-bit compiler only)
-ms!	Compile using small memory model; assume DS! = SS (16-bit compiler only. Note: there is no space between the -ms and the !)
-N	Check for stack overflow
-O	Optimize jumps
-ofilename	Compile .OBJ to "filename" (command-line compiler only)
-O1	Generate smallest possible code
-O2	Generate fastest possible code
-Oa	Optimize assuming pointer expressions are not aliased on common subexpression evaluation
-Ob	Eliminate dead code
-Oc	Eliminate duplicate expressions within basic blocks
-Od	Disable all optimizations
-Oe	Allocate global registers and analyze variable live ranges
-Og	Eliminate duplicate expressions within functions
-OI	Optimize across function boundaries (Intel compiler only)
-Oi	Expand common intrinsic functions
-Ol	Compact loops
-OM	Cache hit optimizations (Intel compiler only)
-Om	Move invariant code out of loops
-Op	Propagate copies
-OS	Pentium instruction scheduling
-Ov	Enable loop induction variable and strength reduction
-P	Force C++ compile (command-line compiler only)

-p	Use Pascal calling convention
-pc	Use C calling convention (Default: -pc, -p-)
-po	Use fastthis calling convention for passing this parameter in registers
-pr	Use fastcall calling convention for passing parameters in registers
-ps	Use stdcall calling convention (32-bit compiler only)
-R	Include browser information in generated .OBJ files
-r	Use register variables (Default)
-rd	Allow only declared register variables to be kept in registers
-RT	Enable run-time type information (Default)
-S	Compile to assembler (command-line compiler only)
-Tx	Specify assembler option “x” (command-line compiler only)
-tD	Compile to a 16-bit real-mode .EXE file (same as -tDe) (command-line compiler only)
-tWM	Generate a 32-bit multi-threaded target (command-line compiler only)
-Uname	Undefine any previous definitions of “name” (command-line compiler only)
-u	Generate underscores (Default)
-V	Use smart C++ virtual tables (Default)
-v	Turn on source debugging
-V0	External C++ virtual tables
-V1	Public C++ virtual tables
-Va	Pass class arguments by reference to a temporary variable (16-bit compiler only)
-Vb	Make virtual base class pointer same size as ‘this’ pointer of the class (Default, 16-bit compiler only)
-VC	Calling convention mangling compatibility
-Vc	Do not add the hidden members and code to classes with pointers to virtual base class members (16-bit compiler only)
-Vd	for loop variable scoping
-Ve	Zero-length empty base classes
-Vf	Far C++ virtual tables (16-bit compiler only)
-Vh	Treat “far” classes as “huge”
-vi	Control expansion of inline functions
-Vmd	Use the smallest representation for member pointers
-Vmm	Member pointers support multiple inheritance
-Vmp	Honor the declared precision for all member pointer types
-Vms	Member pointers support single inheritance
-Vmv	Member pointers have no restrictions (most general representation) (Default)
-Vo	Enable backward compatibility options (command-line compiler only)
-Vp	Pass the ‘this’ parameter to ‘pascal’ member functions as the first
-Vs	Local C++ virtual tables
-Vt	Place the virtual table pointer after nonstatic data members (16-bit compiler only)
-Vv	‘deep’ virtual bases
-w	Display warnings on
-w“xxx”	Enable “xxx” warning message (Default)

-wamb	Ambiguous operators need parentheses
-wamp	Superfluous & with function
-wasm	Unknown assembler instruction
-waus	'identifier' is assigned a value that is never used (Default)
-wbbf	Bit fields must be signed or unsigned int
-wbei	Initializing 'identifier' with 'identifier' (Default)
-wbig	Hexadecimal value contains more than three digits (Default)
-wccc	Condition is always true OR Condition is always false (Default)
-wcln	Constant is long
-wcpt	Nonportable pointer comparison (Default)
-wdef	Possible use of 'identifier' before definition
-wdpu	Declare type 'type' prior to use in prototype (Default)
-wdup	Redefinition of 'macro' is not identical (Default)
-wdsz	Array size for 'delete' ignored (Default)
-weas	Assigning 'type' to 'enum'
-weff	Code has no effect (Default)
-wias	Array variable 'identifier' is near (Default)
-wext	'identifier' is declared as both external and static (Default)
-which	Handler for '<type1>' Hidden by Previous Handler for '<type2>'
-whid	'function1' hides virtual function 'function2' (Default)
-wibc	Base class 'base1' is inaccessible because also in 'base2' (Default)
-will	Ill-formed pragma (Default)
-winl	Functions containing reserved words are not expanded inline (Default)
-wlin	Temporary used to initialize 'identifier' (Default)
-wlvc	Temporary used for parameter 'parameter' in call to 'function' (Default)
-wmsg	User-defined warnings
-wmpc	Conversion to type fails for members of virtual base class base (Default)
-wmpd	Maximum precision used for member pointer type <type> (Default)
-wnak	Non-ANSI Keyword Used: '<keyword>' (Note: Use of this option is a requirement for ANSI conformance)
-wnci	The constant member 'identifier' is not initialized (Default)
-wnfc	Non-constant function 'ident' called for const object
-wnod	No declaration for function 'function'
-wnst	Use qualified name to access nested type 'type' (Default)
-wntd	Use '> >' for nested templates instead of '>>' (Default)
-wnvf	Non-volatile function <function> called for volatile object (Default)
-wobi	Base initialization without a class name is now obsolete (Default)
-wobs	'ident' is obsolete
-wofp	Style of function definition is now obsolete (Default)
-wovl	Overload is now unnecessary and obsolete (Default)
-wpar	Parameter 'parameter' is never used (Default)
-wpch	Cannot create precompiled header: header (Default)
-wpia	Possibly incorrect assignment (Default)
-wpin	Initialization is only partially bracketed
-wpre	Overloaded prefix operator 'operator' used as a postfix operator

-wpro	Call to function with no prototype (Default)
-wrch	Unreachable code (Default)
-wret	Both return and return of a value used (Default)
-wrng	Constant out of range in comparison (Default)
-wrpt	Nonportable pointer conversion (Default)
-wrvl	Function should return a value (Default)
-wsig	Conversion may lose significant digits
-wstu	Undefined structure 'structure'
-wstv	Structure passed by value
-wsus	Suspicious pointer conversion (Default)
-wucp	Mixing pointers to different 'char' types
-wuse	'identifier' declared but never used
-wvoi	Void functions may not return a value (Default)
-wzdi	Division by zero (Default)
-X	Disable compiler autodependency output (Default: -X- use compiler autodependency output)
-x	Enable exception handling (Default)
-xc	Enable compatible exception handling
-xd	Enable destructor cleanup (Default)
-xf	Enable fast exception prologs
-xp	Enable exception location information
-y	Line numbers on
-Y	Enables 24-bit extended addressing mode
-Z	Enable register load suppression optimization
-zAname	Code class set to "name"
-zBname	BSS class set to "name"
-zCname	Code segment set to "name"
-zDname	BSS segment set to "name"
-zEname	Far data segment set to "name"
-zFname	Far data class set to "name"
-zGname	BSS group set to "name"
-zHname	Far data group set to "name"
-zIname	Constant initialized far data segment set to "name"
-zJname	Constant initialized far data class set to "name"
-zKname	Constant initialized far data group set to "name"
-zPname	Code group set to "name"
-zRname	Data segment set to "name"
-zSname	Data group set to "name"
-zTname	Data class set to "name"
-zVname	Far virtual segment set to "name" (16-bit compiler only)
-zWname	Far virtual class set to "name" (16-bit compiler only)
-zXname	Far BSS segment set to "name"
-zYname	Far BSS class set to "name"
-zZname	Far BSS group set to "name"

Command-line options by function

The Paradigm C++ IDE groups the compiler and linker command-line options into the following categories:

- Compiler
- 16-bit compiler
- 32-bit compiler
- C++ options
- Optimizations
- Messages
- Linker

In addition, there are compiler and linker options that you can set from only the command-line:

Table 3-11
Command-line
only options

Option	Description
Configuration Files	
@filename	Read compiler options from the <i>response file</i> "filename"
Response Files	
+filename	Use alternate <i>configuration file</i> "filename"
Compiler Defines	
-Dname	Define "name" to the null string
-Dname=string	Define "name" to "string"
-Uname	Undefine any previous definitions of "name"
Compiler Code Generation	
-b	Make enums always integer-sized (Default: -b- make enums byte-sized when possible)
-K	Default character type unsigned (Default: -K- default character type signed)
-d	Merge duplicate strings (Default)
-po	Use fastthis calling convention for passing this parameter in registers (16-bit compiler only)
-r	Use register variables (Default)
-rd	Allow only declared register variables to be kept in registers
-Y	Enables 24-bit extended addressing
Compiler Floating Point	
-f-	No floating point
-f	Emulate floating point
-ff	Fast floating point
-fp	Correct Pentium FDIV flaw
Compiler Compiler Output	
-X	Disable compiler autodependency output (Default: -X- use compiler autodependency output)
-u	Generate underscores (Default)
-Fc	Generate COMDEFs (16-bit compiler only)

Compiler|Source

-C	Turn nested comments on (Default: -C- turn nested comments off)
-in	Make significant identifier length to be "n" (Default)
-AT	Use Paradigm C++ keywords (also -A-)
-A	Use only ANSI keywords
-AU	Use only UNIX V keywords
-AK	Use only Kernighan and Ritchie keywords

Compiler|Debugging

-k	Turn on standard stack frame (Default)
-N	Check for stack overflow
-vi	Control expansion of inline functions
-y	Line numbers on
-v	Turn on source debugging
-R	Include browser information in generated .OBJ files

Compiler|Precompiled Headers

-H	Generate and use precompiled headers (Default)
-Hu	Use but do not generate precompiled headers
-Hc	Cache precompiled header
-H=filename	Set the name of the file for precompiled headers
-H'xxx'	Stop precompiling after header file xxx

16-bit Compiler|Processor

-1-	Generate 8086 compatible instructions (Default for 16-bit)
-1	Generate the 80186/286 compatible instructions (16-bit only)
-2	Generate 80286 protected-mode compatible instructions (16-bit compiler only)
-3	Generate 80386 protected-mode compatible instructions (Default for 32-bit)
-4	Generate 80386/80486 protected-mode compatible instructions
-5	Generates Pentium instructions
-a	Align byte (Default: -a- use byte-aligning)
-an	Align to "n" where 1=byte, 2=word (16-bit = 2 bytes), 4=Double word (32-bit only, 4 bytes), 8=Quad word (32-bit only, 8 bytes)

16-bit Compiler|Calling Convention

-pc	Use C calling convention (Default: -pc, -p-)
-p	Use Pascal calling convention
-pr	Use fastcall calling convention for passing parameters in registers

16-bit Compiler|Memory Model

-ms	Compile using small memory model (Default, 16-bit compiler only)
-ms!	Compile using small memory model; assume DS!=SS (16-bit compiler only. Note: there is no space between the -ms and the !)
-mm	Compile using medium memory model (16-bit compiler only)
-mm!	Compile using medium memory model; assume DS!=SS (16-bit compiler only. Note: there is no space between the -mm and the !)
-mc	Compile using compact memory model (16-bit compiler only)
-ml	Compile using large memory model (16-bit compiler only)
-mh	Compile using huge memory model

-Fa	Enable page alignment for far segments
-Fb	Enable Borland C++-compatible far data
-Fs	Assume DS=SS in all memory models (16-bit compiler only)
-dc	Move string literals from data segment to code segment (16-bit compiler only)
-Vf	Far C++ virtual tables (16-bit compiler only)
-h	Uses fast huge pointers
-Ff	Create far variables automatically
-Ff=size	Create far variables automatically; set the threshold to " <i>size</i> " (16-bit compiler only)

16-bit Compiler|Segment Names Data

-zRname	Data segment set to " <i>name</i> "
-zSname	Data group set to " <i>name</i> "
-zTname	Data class set to " <i>name</i> "
-zDname	BSS segment set to " <i>name</i> "
-zGname	BSS group set to " <i>name</i> "
-zBname	BSS class set to " <i>name</i> "

16-bit Compiler|Segment Names Far Data

-zEname	Far data segment set to " <i>name</i> "
-zHname	Constant initialized far data segment set to " <i>name</i> "
-zIname	Constant initialized far data class set to " <i>name</i> "
-zJname	Constant initialized far data group set to " <i>name</i> "
-zKname	Far data group set to " <i>name</i> "
-zFname	Far data class set to " <i>name</i> "
-zVname	Far virtual segment set to " <i>name</i> " (16-bit compiler only)
-zWname	Far virtual class set to " <i>name</i> " (16-bit compiler only)
-zXname	Far BSS segment set to " <i>name</i> "
-zYname	Far BSS class set to " <i>name</i> "
-zZname	Far BSS group set to " <i>name</i> "

16-bit Compiler|Segment Names Code

-zCname	Code segment set to " <i>name</i> "
-zPname	Code group set to " <i>name</i> "
-zAname	Code class set to " <i>name</i> "

32-bit Compiler|Processor

-3	Generate 80386 instructions. (Default for 32-bit)
-4	Generate 80486 instructions
-5	Generate Pentium instructions

32-bit Compiler|Calling Convention

-pc	Use C calling convention (Default: -pc, -p-)
-p	Use Pascal calling convention
-pr	Use fastcall calling convention for passing parameters in registers
-ps	Use stdcall calling convention (32-bit compiler only)

C++ Options|Member Pointer

-Vmp	Honor the declared precision for all member pointer types
-------------	---

-Vmv	Member pointers have no restrictions (most general representation) (Default)
-Vmm	Member pointers support multiple inheritance
-Vms	Member pointers support single inheritance
-Vmd	Use the smallest representation for member pointers

C++ Options|C++ Compatibility

-Vd	for loop variable scoping
-K2	Allow only two character types (signed and unsigned). Char is treated as signed.
-VC	Calling convention mangling compatibility
-Vb	Make virtual base class pointer same size as 'this' pointer of the class (Default, 16-bit compiler only)
-Va	Pass class arguments by reference to a temporary variable (16-bit compiler only)
-Vc	Do not add the hidden members and code to classes with pointers to virtual base class members (16-bit compiler only)
-Vp	Pass the 'this' parameter to 'pascal' member functions as the first
-Vv	'deep' virtual bases
-Vt	Place the virtual table pointer after nonstatic data members (16-bit compiler only)
-Vh	Treat "far" classes as "huge"

C++ Options|Virtual Tables

-V	Use smart C++ virtual tables (Default)
-Vs	Local C++ virtual tables
-V0	External C++ virtual tables
-V1	Public C++ virtual tables

C++ Options|Templates

-Jg	Generate definitions for all template instances and merge duplicates (Default)
-Jgd	Generate public definitions for all template instances; duplicates result in redefinition errors
-Jgx	Generate external references for all template instances

C++ Options|Exception Handling

-x	Enable exception handling (Default)
-xp	Enable exception location information
-xd	Enable destructor cleanup (Default)
-xf	Enable fast exception prologs
-xc	Enable compatible exception handling
-RT	Enable run-time type information (Default)

C++ Options|General

-Ve	Zero-length empty base classes
------------	--------------------------------

Optimizations

-Od	Disable all optimizations
-O1	Generate smallest possible code
-O2	Generate fastest possible code

Optimizations|16- and 32-bit

-Oc	Eliminate duplicate expressions within basic blocks
------------	---

- Og** Eliminate duplicate expressions within functions
- Oi** Expand common intrinsic functions
- Ov** Enable loop induction variable and strength reduction

Optimizations|16-bit

- O** Optimize jumps
- Ol** Compact loops
- Z** Enable register load suppression optimization
- Ob** Eliminate dead code
- OW** Suppress the inc bp/dec bp on windows far functions (16-bit compiler only)
- Oe** Allocate global registers and analyze variable live ranges
- Oa** Optimize assuming pointer expressions are not aliased on common subexpression evaluation
- Om** Move invariant code out of loops
- Op** Propagate copies

Optimizations|32-bit

- OS** Pentium instruction scheduling
- OM** Cache hit optimizations (Intel compiler only)
- OI** Optimize across function boundaries (Intel compiler only)

Messages

- w** Display warnings on
- wxxx** Enable "xxx" warning message (Default)
- gn** Warnings: stop after "n" messages (Default: 255)
- jn** Errors: stop after "n" messages (Default)

Messages|Portability

- wrpt** Nonportable pointer conversion (Default)
- wcpt** Nonportable pointer comparison (Default)
- wrng** Constant out of range in comparison (Default)
- wcln** Constant is long
- wsig** Conversion may lose significant digits
- wucp** Mixing pointers to different 'char' types

Messages|ANSI Violations

- wvoi** Void functions may not return a value (Default)
- wret** Both return and return of a value used (Default)
- wsus** Suspicious pointer conversion (Default)
- wstu** Undefined structure 'structure'
- wdup** Redefinition of 'macro' is not identical (Default)
- wbig** Hexadecimal value contains more than three digits (Default)
- wbbf** Bit fields must be signed or unsigned int
- wext** 'identifier' is declared as both external and static (Default)
- wdpu** Declare type 'type' prior to use in prototype (Default)
- wzdi** Division by zero (Default)
- wbei** Initializing 'identifier' with 'identifier' (Default)
- wpin** Initialization is only partially bracketed
- wnak** Non-ANSI Keyword Used: '<keyword>' (Note: Use of this option is a requirement for ANSI conformance)

Messages|Obsolete C++

-wobi	Base initialization without a class name is now obsolete (Default)
-wofp	Style of function definition is now obsolete (Default)
-wpre	Overloaded prefix operator 'operator' used as a postfix operator
-wovl	Overload is now unnecessary and obsolete (Default)

Messages|Potential C++ Errors

-wnci	The constant member 'identifier' is not initialized (Default)
-weas	Assigning 'type' to 'enum'
-whid	'function1' hides virtual function 'function2' (Default)
-wnfc	Non-constant function 'ident' called for const object
-wibc	Base class 'base1' is inaccessible because also in 'base2' (Default)
-wdsz	Array size for 'delete' ignored (Default)
-wnst	Use qualified name to access nested type 'type' (Default)
-which	Handler for '<type1>' Hidden by Previous Handler for '<type2>'
-wmpc	Conversion to type fails for members of virtual base class base (Default)
-wmpd	Maximum precision used for member pointer type <type> (Default)
-wntd	Use '> >' for nested templates instead of '>>' (Default)
-wnvf	Non-volatile function <function> called for volatile object (Default)

Messages|Inefficient C++ Coding

-winl	Functions containing reserved words are not expanded inline (Default)
-wlin	Temporary used to initialize 'identifier' (Default)
-wlvc	Temporary used for parameter 'parameter' in call to 'function' (Default)

Messages|Potential Errors

-wpia	Possibly incorrect assignment (Default)
-wdef	Possible use of 'identifier' before definition
-wnod	No declaration for function 'function'
-wpro	Call to function with no prototype (Default)
-wrvl	Function should return a value (Default)
-wamb	Ambiguous operators need parentheses
-wccc	Condition is always true OR Condition is always false (Default)

Messages|Inefficient Coding

-waus	'identifier' is assigned a value that is never used (Default)
-wpar	Parameter 'parameter' is never used (Default)
-wuse	'identifier' declared but never used
-wstv	Structure passed by value
-wrch	Unreachable code (Default)
-weff	Code has no effect (Default)

Messages|General

-wasn	Unknown assembler instruction
-will	Ill-formed pragma (Default)
-wias	Array variable 'identifier' is near (Default)
-wamp	Superfluous & with function
-wobs	'ident' is obsolete
-wpch	Cannot create precompiled header: header (Default)
-wmsg	User-defined warnings

Linker options

General

Map file

16-bit linker

16-bit optimizations

32-bit linker

Warnings

Command-line only options

16- and 32-bit command-line options

Linker supported command-line options

32-bit command-line options

Browsing through your code

The browser lets you search through your object hierarchies, classes, functions, variables, types, constants, and labels that your program uses. The browser also lets you:

- Graphically view the hierarchies in your application, then select the object of your choice and view the functions and symbols it contains.
- List the variables your program uses, then select one and view its declaration, list all references to it in your program, or go to where it is declared in your source code.
- List all the classes your program uses, then select one and list all the symbols in its interface part. From this list, you can select a symbol and browse as you would with any other symbol in your program.

Using the browser

If the program in the current Edit window or the first file in your project has not yet been compiled, the Paradigm C++ IDE must first compile your program before invoking the browser.

If you try to browse a variable or class definition (or any symbol that does not have symbolic debug information), the Paradigm C++ IDE displays an error message.

If you changed the following default settings on the Project options dialog box, before you use the browser, be sure to:

1. Choose Options|Project.
2. Choose Compiler|Debugging and check
 - Debug information in OBJs
 - Browser reference information in OBJs
3. Choose Linker|General and check Include debug information.
4. Compile your application.

Starting the browser

To start browsing through your code, choose one of the following menu or SpeedBar commands: From the main menu or the SpeedBar:

- Search|Browse symbol
- Search|Browse Classes
- Search|Browse Globals

Browser views

The browser provides the following views:

- Global symbols
- Objects (Class overview)
- Symbol declaration

- Class inspection
- References

Browsing objects (class overview)

Choose Search|Browse classes to see an overall view of the object hierarchies in your application, as well as the small details.

The browser draws your objects and shows their ancestor-descendant relationships in a horizontal tree. The red lines in the hierarchy help you see the immediate ancestor-descendant relationships of the currently selected object more clearly.

To see more detail about a particular object, double-click it. (If you are not using a mouse, select the object by using your arrow cursor keys and press *Enter*.) The browser lists the symbols (the procedures, functions, variables, and so on) used in the object.

One or more letters appear to the left of each symbol in the object that describe what kind of symbol it is. "See Browser filters and letter symbols".

Browsing global symbols

Choose Search|Browse globals to open a window that lists every global symbol in your application in alphabetical order.

To see the declaration of a particular symbol listed in the browser, use one of the following methods:

- Double-click the symbol
- Select the symbol and press *Enter*
- Select the symbol, choose Browse symbol from the SpeedMenu

Search

The Search input box at the bottom of the window lets you quickly search through the list of global symbols by typing the first few letters of the symbol name. As you type, the highlight bar in the list box moves to a symbol that matches the typed characters.

Browser SpeedMenu

Once you select the global symbol you are interested in, you can use the following commands on the Browser SpeedMenu:

- Edit Source
- Browse Symbol
- Browse References
- Return to Previous View
- Print Class Hierarchy
- Toggle Window Mode

Browsing symbols in your code

You can browse any symbol in your code without viewing object hierarchies or lists of symbols first.

To do so, highlight or place the insertion point on the symbol in your code and choose Browse symbol. from the Search menu or the Edit window SpeedMenu.

If the symbol you select is a structured type, the browser shows you all the symbols in the scope of that type. You can then choose to inspect any of these further. For example, if you choose an object type, you will see all the symbols listed that are within the scope of the object.

Symbol declaration window

This Browser window shows the declaration of the selected symbol.

You can use the following commands on the Browser SpeedMenu:

- Edit Source
- Browse References
- Browse Class Hierarchy
- Return to Previous View
- Toggle Window Mode

Browsing references

This Browser window shows the references to the selected symbol.

You can use the following commands on the Browser SpeedMenu:

- Edit Source
- Browse Class Hierarchy
- Return to Previous View
- Toggle Window Mode
- Set Options

Class inspection window

This Browser window shows the symbols (functions and variables) used in the selected class.

Once you select the symbol you are interested in, you can use the following commands on the Browser SpeedMenu:

- Edit Source
- Browse Symbol
- Browse References
- Browse Class Hierarchy
- Return to Previous View
- Toggle Window Mode
- Set Options

Browser filters and letter symbols

When you browse a particular symbol, the same letters that appear on the left that identify the symbol appear in a Filters matrix at the bottom of the Browser window. The Filters matrix has a column for each letter which can appear in the top or bottom row of the column.

Use the filters to select the type of symbols you want to see listed. (You can also use the Browser options settings to specify the types of symbols you want to see listed.)

Table 4-1
Browser letter
symbols

Letter	Symbol
F	Function
T	Type
V	Variable
C	Integral constants
?	Debuggable
I	Inherited from an ancestor
v	Virtual method



In some cases, more than one letter appears next to a symbol. Additional letters appear to the right of the letter identifying the type of symbol and further describe the symbol:

To view all instances of a particular type of symbol

Click the top cell
of the column.

For example, to view all the variables in the currently selected object, click the top cell in the V column. All the variables used in the object appear.

To hide all instances of a particular type of symbol

Click the bottom
cell of the letter
column.

For example, to view only the functions and procedures in an object, you need to hide all the variables. Click the bottom cell in the V column, and click the top cells in the F and P columns.

To change several filter settings at once

Drag your mouse over the cells you want to select in the Filters matrix.

Customizing the browser

Use the Environment Options dialog box to select the Browser options you want to use.

1. Choose Options|Environment.
2. Choose Browser.
3. Specify the types of symbols you want to have visible in the Browser using the Visible symbols option.
4. Specify how many browser views you can have open at one time. See single or multiple Browser window mode in the Browser window behavior option.

Using the integrated debugger

No matter how careful you are when you code, your program is likely to have errors or *bugs* that prevent it from running the way you intended. *Debugging* is the process of locating and fixing the errors in your program.

The Paradigm C++ IDE contains an *integrated debugger* that lets you debug 16- and 32-bit embedded applications without leaving the development environment. Among other things, the integrated debugger lets you control the execution of your program, inspect the values of variables and items in data structures, modify the values of data items while debugging. You can access the functionality of the integrated debugger through two menus: Debug and View along with local menus and keystrokes. This chapter introduces you to the functionality of the integrated debugger and gives a brief overview of the debugging process.

Types of bugs

The integrated debugger can help find two basic types of programming errors: run-time errors and logic errors.

Run-time errors

If your program successfully compiles, but fails when you run it, you've encountered a *run-time error*. Your program contains valid statements, but the statements cause errors when they're executed. For example, your program might be trying to open a nonexistent file, or might be trying to divide a number by zero. The operating system detects run-time errors and stops your program execution if such an error is encountered.

Without a debugger, run-time errors can be difficult to locate because the compiler doesn't tell you where the error is located in your source code. Often, the only clue you have to work with is where your program failed and the error message generated by the run-time error.

Although you can find run-time errors by searching through your program source code, the integrated debugger can help you quickly track down these types of errors. Using the integrated debugger, you can run to a specific program location. From there, you can begin executing your program one statement at a time, watching the behavior of your program with each step. When you execute the statement that causes your program to fail, you have pinpointed the error. From there, you can fix the source code recompile the program, and resume testing your program.

Logic errors

Logic errors are errors in design and implementation of your program. Your program statements are valid (they do *something*), but the actions they perform are not the actions you had in mind when you wrote the code. For instance, logic errors can occur when variables contain incorrect values, or when the output of your program is incorrect.

Logic errors are often the most difficult type of errors to find because they can show up in places you might not expect. To be sure your program works as designed, you must thoroughly test all of its aspects. Only by scrutinizing each portion of the user interface and output of your program can you be sure that its behavior corresponds to its design. As with run-time errors, the integrated debugger helps you locate logic errors by letting you monitor the values of your program variables and data objects as your program executes.

Planning a debugging strategy

After program design, program development consists of a continuous cycle of program coding and debugging. Only after you thoroughly test your program should you distribute it to your end users. To ensure that you test all aspects of your program, it's best to have a thorough plan for your debugging cycles.

One good debugging method involves breaking your program down into different sections that you can systematically debug. By closely monitoring the statements in each program section, you can verify that each area is performing as designed. If you do find a programming error, you can correct the problem in your source code, recompile the program, and then resume testing.

Starting a debugging session

To start a debugging session:

1. Build your program with debug information.
2. Run your program from within the Paradigm C++ IDE.

When debugging, you have complete control of your program's execution. You can pause the program at any point to examine the values of program variables and data structures, to view the sequence of function calls, and to modify the values of program variables to see how different values affect the behavior of your program.

Compiling with debug information

Before you can begin a debugging session, you must compile your program with *symbolic debug information*. Symbolic debug information, contained in a *symbol table*, enables the debugger to make connections between your program's source code and the machine code that's generated by the compiler. This lets you view the actual source code of your program while running the program through the debugger.

To generate symbolic debug information for your project:

1. In the Project window, select the project node.
2. Choose Options|Project to open the Project Options dialog box.
3. From the Compiler|Debugging topic, check Debug Information in .OBJs to include debug information in your project .OBJ files (this option is checked by default).
4. From the Linker|General topic, check Include Debug Information. This option transfers the symbolic debug information contained in your .OBJ files to the .ROM file (this option is checked by default).

Adding debugging information to your files increases their file size. Because of this, you'll want to include debug information in your files only during the development stage of your project. Once your program is fully debugged, compile your program without debug information to reduce the final .AXE file size.



Not all .OBJ files in your project need symbolic debug information - only those modules you need to debug must contain a symbol table. However, since you can't statement step into a module that doesn't contain debug information, it's best to compile all your modules with a minimum of line number debug information during the development stages of your project.

Running your program in the Paradigm C++ IDE

Once you've compiled your program with debug information, you can begin a debugging session by running your program in the Paradigm C++ IDE. By running your program in the Paradigm C++ IDE, you have control of when the program runs and when it pauses. Whenever the program is paused in the Paradigm C++ IDE, the debugger takes control.

When your program is running under the Paradigm C++ IDE, it behaves as it normally would: your program creates windows, accepts user input, calculates values, and displays output. During the time that your program is not running, the debugger has control, and you can use its features to examine the current state of the program. By viewing the values of variables, the functions on the call stack, and the program output, you can ensure that the area of code you're examining is performing as it was designed.

As you run your program through the debugger, you can watch the behavior of your application in the windows it creates. For best results during your debugging sessions, arrange your screen so you can see both the Paradigm C++ IDE Edit window and your application window as you debug. To keep windows from flickering as the focus alternates between the debugger windows and those of your application, arrange the windows so they don't overlap (tile the windows). With this setup, your program's execution will be quicker and smoother during the debugging session.

Specifying program arguments

If the program you want to debug uses command-line arguments, you can specify those arguments in the Paradigm C++ IDE in two ways.

First:

1. Choose Options|Environment then select the Debugger topic.
2. In the Arguments text box, type the arguments you want to use when you run your program under the control of the integrated debugger.

Second:

1. Choose Debug|Load.
2. Type your program name and arguments in the Load dialog box.

Controlling program execution

An important advantage of a debugger is that it lets you control the execution of your program; you can control whether your program will execute a single machine instruction, a single line of code, an entire function, or an entire program block. By dictating when the program should run and when it should pause, you can quickly move over the sections that you know work correctly and concentrate on the sections that are causing problems.

The integrated debugger lets you control the execution of your program in the following ways:

- Running to the cursor location
- Stepping through code

- Running to a breakpoint
- Pausing your program

When running code through the debugger, program execution can be based on lines of source code or on machine instructions. When debugging at the source level, the integrated debugger lets you control the rate of debugging to the level of a single line of code. However, the debugger considers multiple program statements on one line of text to be a single line of code; you cannot individually debug multiple statements contained on a single line of text. In addition, the debugger regards a single statement that's spread over several lines of text as a single line of code.

Running to the cursor location

Often when you start a debugging session, you'll want to run your program to a spot just before the suspected location of the problem. At that point, use the debugger to ensure that all data values are as they should be. If everything is OK, you can run your program to another location, and again check to ensure that your program is behaving as it should.

To run to a specific source line:

1. In the Edit window or CPU window, position the cursor on the line of code where you want to begin (or resume) debugging.
2. Run to the cursor location in one of the following ways:
 - Click the Run To Here button on the SpeedBar.
 - Choose Run To Current from the Edit window SpeedMenu
 - Choose Run To Current in the Disassembly pane of the CPU window.

To run to a specific machine instruction:

1. After your process is loaded, open a CPU view and position the disassembly pane so that the highlight is on the address to which you want to run.
2. Choose Run To Current from the disassembly pane SpeedMenu, or click the Run To Here button on the SpeedMenu.

When you run to the cursor, your program executes at full speed until the execution reaches the location marked by the cursor in the Edit window, or highlight in the CPU window. When the execution encounters the code marked by the text cursor or highlighted, the debugger regains control and places the execution point on that line of code.

The execution point

The *execution point* marks the next line of source code to be executed by the debugger. Whenever you pause your program execution within the debugger (for example, whenever you run to the cursor or step to a program location), the debugger highlights a line of code using a green arrow and colored background (depending on your color setup), marking the location of the execution point.

The execution point always shows the next line of code to be executed, whether you are going to step through, step into, or run your program at full speed. If there is no source associated with the code at the current execution point, a CPU window is opened showing the instruction with the instruction at the current execution point.

Finding the execution point

While debugging, you're free to open, close, and navigate through any file in an Edit window. Because of this, it's easy to lose track of the next program statement to execute, or the location of the current program scope. To quickly return to the execution point, choose Debug|Source At Execution Point or click the SpeedBar button. Even if you've closed the Edit window containing the execution point, Find Execution Point opens an Edit window, and highlights the source code containing the execution point.

If there is no source associated with the code at the current execution point, you will get an error stating that no line corresponds to the address. If this happens, you can see the current execution point by opening the CPU window.

Stepping through code

Stepping is the simplest way to move through your code one statement at a time. Stepping lets you run your program one line (or instruction) at a time – the next line of code (or instruction) will not execute until you tell the debugger to continue. After each step, you can examine the state of the program, view the program output, and modify program data values. Then, when you are ready, you can continue executing the next program statement.

There are two basic ways to step through your code:

- Step Into** The Step Into command is available on the SpeedMenu in the Edit window or by using *F8*. Step Into causes the debugger to walk through your code one statement at a time. If the *execution point* is located on a function call, the debugger moves to the first line of code that defines that function. From here, you can execute that function, one statement at a time. When you step past the return of the function, the debugger resumes stepping from the point where the function was called. Using the Step Into command to step through your program one statement at a time is known as single stepping.
- Step Over** The Step Over command is also available on the SpeedMenu in the Edit window or by using *F7*. Step Over is the same as Step Into, except that if you issue the Step Over command when the execution point is on a function call, the debugger executes the function at full speed, and pauses the execution on the line of code following the function call.

Stepping rules

The debugger steps over single lines of lines of code based on the following rules:

- If you string several statements together on one line, you cannot debug those statements individually; the debugger treats all statements as a single line of code.
- If you spread a single statement over multiple lines in your source file, the debugger executes all the lines as a single statement.



To ensure that the debugger accurately represents your C++ source code while stepping, choose Options|Project|Compiler|Debugging and click Out-of-Line Inline Functions.

Stepping into

To Step Into code, choose Statement|Step Into from the Edit window SpeedMenu or press *F7* (default keyboard mapping).

When you choose Step Into, the debugger executes the code highlighted by the *execution point*. If the execution point is highlighting a function call, the debugger moves the execution point to the first line of code that defines the function being called.

If the executing statement calls a function that does not contain debug information, the debugger opens the CPU window and positions the execution point on the disassembled instruction that corresponds to the function definition in memory.

Example

The following code fragment shows how Step Into works. Suppose these two functions are in a program that was compiled with debug information:

```
func_1() {
    statement_a;
    func_2();
    statement_b;
}

func_2() {
    int customers;
    statement_m;
}
```

If you choose Step Into when the execution point is on `statement_a` in `func_1`, the execution point moves to highlight the call to `func_2`. Choosing Step Into again positions the execution point at the first line in the definition of `func_2`. Another Step Into command moves the execution point to `statement_m`, the first executable line of code in `func_2`.

When you step past a function return statement (in this case, the closing function brace), the debugger positions the execution point on the line following the original function call. Here, the debugger would highlight `statement_b` with the execution point.

As you debug, you can choose to Step Into some functions and Step Over others. Use Step Into when you need to fully test the function highlighted by the execution point.

Stepping over

To Step Over code, choose Statement|Step Over from the Edit Window SpeedMenu or press *F8* (default keyboard mapping).

When you choose the Step Over command, the debugger executes the code highlighted by the *execution point*. If the execution point is highlighting a function call, the debugger executes that function at full speed, including any function calls within the function highlighted by the execution point. The execution point then moves to the next complete line of code.

Example

The following code fragment shows how Step Over works. Suppose these two functions are in a program that was compiled with debug information:

```
func_1() {
    statement_a;
    func_2();
    statement_b;
}

func_2() {
    statement_m;
    func_3();
}
```

If you choose Step Over when the execution point is on `statement a` in `func 1`, the execution point moves to highlight the call to `func 2`. Choosing Step Over again runs `func 2` at full speed, moving the execution point to `statement b`. Notice that when you step over `func 2`, `func 3` is also run at full speed.

As you debug, you can choose to Step Into some functions and Step Over others. Step Over is good to use when you have fully tested a function, and you do not need to single step through its code.

Debugging member functions and external code

If you use classes in your programs, you can still use the integrated debugger to *step* through the member functions in your code. The debugger handles member functions the same way it would step through functions in a program that is not object-oriented.



If you define a member function inline, then you should check Out-of-line inline functions to facilitate debugging the inline function.

You can also step through or step over external code written in any language (including C, C++, Object Pascal, and assembly language) as long as the code meets all the requirements for external linking and contains full Paradigm symbolic debugging information. If the external code does not contain Paradigm debug information, you can still step through the code using the CPU window.

Running to a breakpoint

You set *breakpoints* on lines of source code where you want the program execution to pause during a run. Running to a breakpoint is similar to running to a cursor position that the program runs at full speed until it reaches a certain source-code location. However, unlike Run to Cursor, you can have multiple breakpoints in your code and you can customize each one so it pauses the program's execution only when a specified condition is met. For more information on breakpoints, see “Examining program data values,” page 5-148.

Pausing a program

In addition to stepping over or through code, you can also pause your program while it is running. Choosing Debug|Pause Process causes the debugger to pause your program. You can use the debugger to examine the value of variables and inspect data at this state of the program. When you are done, choose Debug|Run to continue the execution of your program.

Terminating the program

Sometimes while debugging, you will find it necessary to restart the program from the beginning. For example, you might need to restart the program if you step past the location of a bug, or if variables or data structures become corrupted with unwanted values.

Choose Debug|Terminate debug session (or press *Ctrl-F2*) to end the current program run. Terminating a program closes all open program files, releases all memory allocated by the program, and clears all variable settings. However, terminating a program does not delete any breakpoints or watches that you might have set. This makes it easy to resume a debugging session.

Using breakpoints

You use *breakpoints* is similar to using the Run to Cursor command in that the program runs at full speed until it reaches a certain point. But, unlike Run to Cursor, you can have multiple breakpoints and you can choose to stop at a breakpoint only under certain conditions. Once your program's execution is paused, you can use the debugger to examine the state of your program.

The Paradigm C++ IDE keeps track of all your breakpoints during a debugging session and associates them with your current project. You can maintain all your breakpoints from a single Breakpoints window and not have to search through your source code files to look for them.

Debugging with breakpoints

When you run your program from the Paradigm C++ IDE, it will stop whenever the debugger reaches the location in your program where the breakpoint is set, but before it executes the line or instruction. The line that contains the breakpoint (or the line that most closely corresponds to the program location where the breakpoint is set) appears in the Edit window highlighted by the *execution point*. At this point, you can perform any other debugging actions.

Setting breakpoints

You can set a breakpoint the following ways:

To set an unconditional breakpoint on a line in your source code, use one of the following methods:

- Place the insertion point on a line in an Edit window and choose Toggle|Breakpoint from the Edit window SpeedMenu. or press *F5* (default keyboard setting).
- Click the *gutter* in an Edit window next to the line where you want to set a breakpoint.

Setting an unconditional breakpoint

To set an unconditional breakpoint on a machine instruction:

1. Highlight a machine instruction in the Disassembly pane in the CPU window.
2. Choose Toggle Breakpoint on the SpeedMenu or press *F5* (default keyboard setting).

Setting a conditional breakpoint

To set a conditional breakpoint on a line or machine instruction:

1. Place the insertion point on a line in an Edit window or highlight a line in the Disassembly pane of the CPU window.
2. Choose Debug|Add Breakpoint or choose Add Breakpoint from the SpeedMenu.
3. Complete the information on the Add Breakpoint dialog box.
4. Do one of the following:
 - Click the Advanced button to display the Breakpoint Condition/Action Options dialog box.
 - Supply the conditions and action settings you want. See "Creating conditional breakpoints," page 5-137.
 - Specify option *set* in the Options input box.

Setting other breakpoints

To set other types of breakpoints:

1. Choose Debug|Add Breakpoint (or press *F5* in the default keyboard setting) from anywhere in the Paradigm C++ IDE or choose Add Breakpoint from the SpeedMenu in an active Edit or Breakpoint window, or the Disassembly pane of the CPU window.
2. Select a breakpoint type on the Add Breakpoint dialog box and supply any additional information associated with the type of breakpoint selected.
3. Either
 - Click OK to set an unconditional breakpoint.
 - Click the Advanced button to display the Breakpoint Condition/Action Options dialog box. See “Creating conditional breakpoints,” page 5-137.

To view a breakpoint

Choose View|Breakpoint to display the Breakpoints window.

Setting breakpoints after program execution begins

While your program is running, you can switch to the debugger (just like you switch to any Windows application) and set a breakpoint. When you return to your application, the new breakpoint is set, and your application will pause or perform a specified action when it reaches the breakpoint.

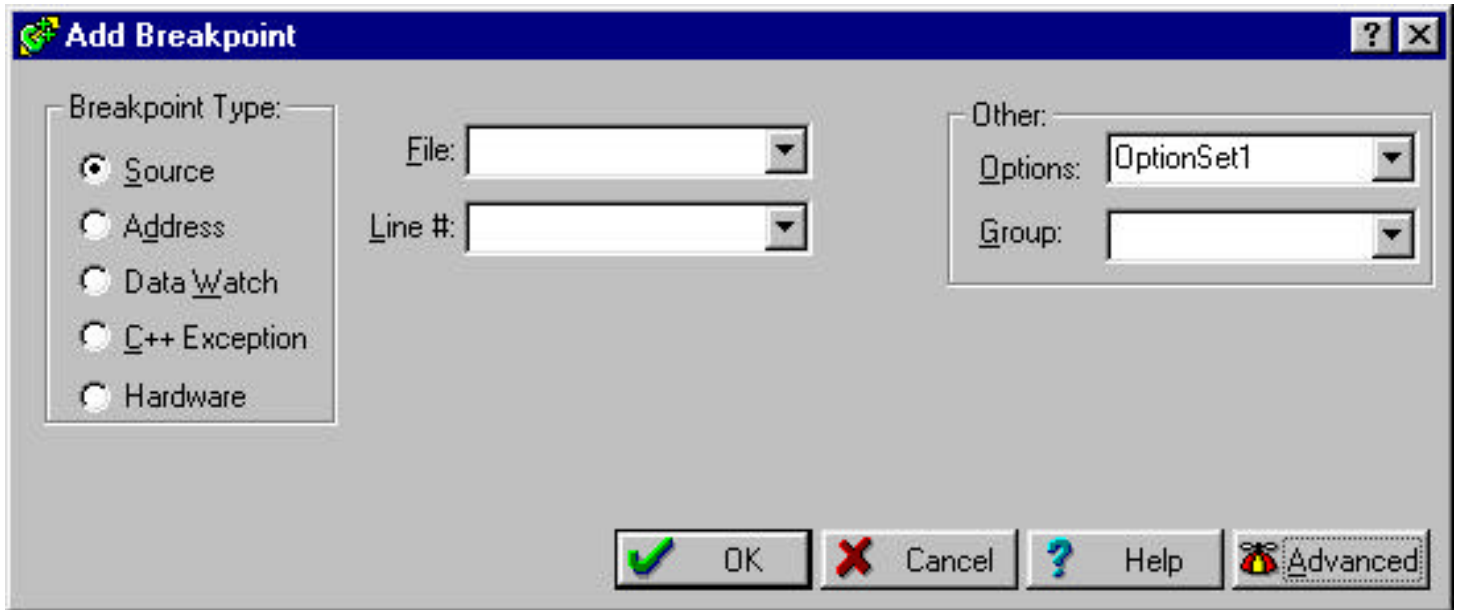
Creating conditional breakpoints

Use a *conditional breakpoint* when you want the debugger to activate a breakpoint only under certain conditions. For example, you may not want a breakpoint to activate every time it is encountered, especially if the line containing the breakpoint is executed many times before the actual occurrence in which you are interested. Likewise, you may not always want a breakpoint to pause program execution. In these cases, use a conditional breakpoint.

To set a conditional breakpoint:

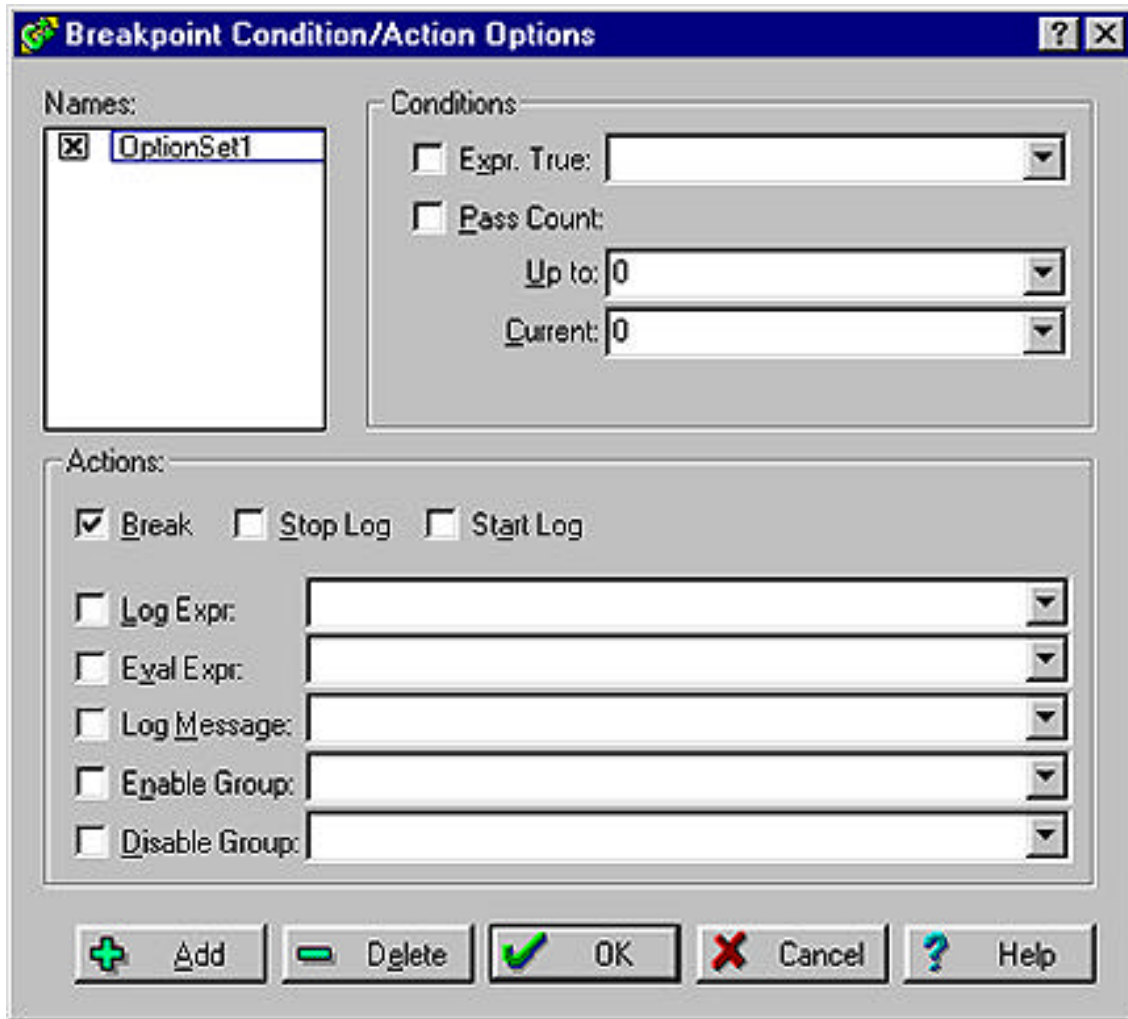
1. Choose Debug|Add Breakpoint to open the Add Breakpoint dialog box.

Figure 5-1 Add Breakpoint dialog box



2. Select a *breakpoint type* and supply the applicable information.
3. Click Advanced to display the Breakpoint Condition/Action Options dialog box.
4. Click Expr. True and enter an expression that tells the debugger when to trigger the breakpoint. If the condition is not met, the debugger ignores the breakpoint along with any of its actions.
5. If you want the debugger to activate a breakpoint only after it has been reached a certain number of times, click Pass count and enter the number of passes. Otherwise, your program will pause every time the breakpoint is activated.
6. If you want program execution to pause when the breakpoint is activated, click **Break** (the default). Otherwise, your program will not pause when the debugger activates the breakpoint.
7. If you want the debugger to perform various actions when the breakpoint activates, use the Actions settings. Otherwise, click OK.

Figure 5-2 Breakpoint Condition/Action Options dialog box



Removing breakpoints

You can remove a breakpoint the following ways:

From an Edit window

Double-click the gutter in an Edit window next to the line that contains the breakpoint you want to remove.

From an Edit window or the Disassembly pane of the CPU window

1. Place the insertion point on the line or highlight the instruction where the breakpoint is set.
2. Choose Toggle Breakpoint from the SpeedMenu.

From the Breakpoints window

1. Choose View|Breakpoint to display the Breakpoints window.
2. Select one or more breakpoints.
3. Choose Remove Breakpoint(s) from the SpeedMenu.



To select multiple breakpoints in the Breakpoints window, hold down the Shift or Ctrl key as you select each breakpoint.

Disabling and enabling breakpoints

Disable a breakpoint when you prefer not to activate it the next time you run your program, but want to save it for later use. The breakpoint remains listed in the Breakpoints window and available for you to enable when you want.

To enable or disable a breakpoint

1. Choose View|Breakpoint to open the Breakpoints window.
2. Click the checkbox next to the breakpoint to enable it or clear the checkbox to disable it.

To disable or enable selected breakpoints

1. In the Breakpoints window, hold down the *Shift* or *Ctrl* key as you select each breakpoint.
2. Choose Enable/Disable Breakpoints from the SpeedMenu.

To use a breakpoint to disable or enable a group of breakpoints

1. Choose Debug|Add Breakpoint to open the Add Breakpoint dialog box.
2. Click Options to open the Breakpoint Condition/Action Options dialog box.
3. Click Enable Group or Disable Group and enter a group name.

Viewing and editing code at a breakpoint

Even if a breakpoint is not in your current Edit window, you can quickly locate it in your source code.

Viewing code at a breakpoint

1. Choose View|Breakpoint to display the Breakpoints window.
2. Select a breakpoint.
3. Choose View Source on the Breakpoints window SpeedMenu.

The source code displays in an Edit window at the breakpoint line and the Breakpoints window remains active. If the source code is not currently open in an Edit window, the Paradigm C++ IDE opens a new Edit window.

Editing code at a breakpoint

1. Choose View|Breakpoint to display the Breakpoints window.
2. Select a breakpoint.
3. Choose Edit Source from the Breakpoints window SpeedMenu.

The source code displays in an active Edit window with your cursor positioned on the breakpoint line, ready for you to edit. If the source code is not currently open in an Edit window, the Paradigm C++ IDE opens a new Edit window.

Resetting invalid breakpoints

A breakpoint must be set on executable code; otherwise, it is invalid. For example, a breakpoint set on a comment, a blank line, or a declaration is invalid. A common error is to set a breakpoint on code that is conditionalized out using `#if` or `#ifdef`.

If you set an invalid breakpoint and run your program, the debugger displays an Invalid Breakpoint dialog box.

To reset an invalid breakpoint

1. Close the Invalid Breakpoint dialog box.
2. Open the Breakpoints window.
3. Find the invalid breakpoint and delete it.
4. Set the breakpoint in a proper location and continue to run your program.



If you ignore the Invalid Breakpoint (by dismissing the dialog box) and then choose Run, the Paradigm C++ IDE executes your program, but does not enable the invalid breakpoint.

Using breakpoint groups

To remove a breakpoint from a group, select the group name and press Delete.

The integrated debugger lets you group breakpoints together so you can enable or disable them with a single breakpoint action.

Creating a breakpoint group

1. Choose Debug|Add Breakpoint to open the Add Breakpoint dialog box.
2. Enter a name in the Group input box.

Disabling or enabling a breakpoint group

1. Choose Debug|Add Breakpoint to open the Add Breakpoint dialog box.
2. Click Options to open the Breakpoint Condition/Action Options dialog box.
3. Click Enable Group or Disable Group and enter a group name.

Using breakpoint option sets

You can also create an option set when you create or edit a breakpoint.

To quickly specify the behavior of one more breakpoints as you create or modify them, store breakpoint settings in an *option set*.

Creating a breakpoint option set

1. Choose Debug|Breakpoint options to open the Breakpoint Condition/Action Options dialog box.
2. Enter the conditions and actions. See “Creating conditional breakpoints,” page 5-137.
3. Click Add.
4. Enter a name in the dialog box that displays and click OK.

Associating a breakpoint with an option set

- Enter an Option name in the Add or Edit Breakpoints dialog box.

Deleting an option set

1. Choose Debug|Breakpoint options to open the Breakpoint Condition/Action Options dialog box.

2. Select an Option set and click Delete.

Changing breakpoint options

To change the conditions and actions of a breakpoint:

1. Choose View|Breakpoint to open the Breakpoints window.
2. Double-click on a breakpoint or choose Edit Breakpoint from the SpeedMenu.
3. Change the option set in the Options input box on the Edit Breakpoint dialog box.

or

Add new information as described in “Creating conditional breakpoints,” page 5-137.

Changing the color of breakpoint lines

To use colors to indicate if a breakpoint is enabled, disabled, or invalid:

1. Choose Options|Environment.
2. Select Syntax Highlighting and choose Customize.
3. From the Element list, select the following breakpoint options you want to change:
 - Enabled Break
 - Disabled Break
 - Invalid Break
4. Select the background (BG) and foreground (FG) colors you want.
5. If you want highlighting, choose Default Color.

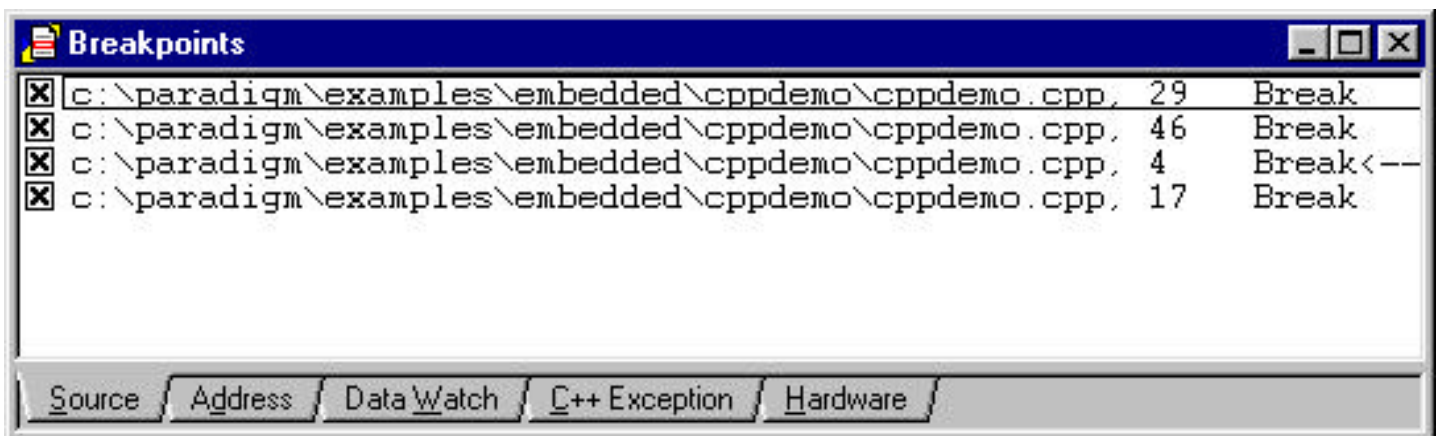
Using the Breakpoints window

The Breakpoints window lists all breakpoints currently set in the loaded project (or the file in the active Edit window if a project is not loaded) and contains a tab for each of the following *breakpoint types*.



To display the Breakpoints window, choose View|Breakpoint (Figure 5-3)

Figure 5-3 Breakpoints window



The Breakpoints window lets you perform the following actions:

Choose a
command from
the Breakpoint
window
SpeedMenu.

- Click the checkbox beside a breakpoint to enable it or clear the checkbox to disable the breakpoint.
- Double-click on a breakpoint or press *Enter* to open the Edit Breakpoint dialog box to change breakpoint settings.

About the Breakpoints window

The Breakpoints window provides the following information about each breakpoint:

- Name of the source code file in which the breakpoint is set (for source breakpoints).
- Location (such as line number, file name, module, or address number) where the breakpoint is set.
- Current state of the breakpoint:
 - Verified** - The breakpoint is legal and validated when the process was loaded.
 - Unverified** - The process has not been loaded since you added the breakpoint.
 - Invalid** - The breakpoint is illegal. The line on which you set the breakpoint does not contain executable code (such as a blank line, comment, or declaration) and the debugger will ignore it.
- Number of times the debugger must reach the breakpoint before activating the breakpoint. This information appears after a breakpoint has been activated. See “Pass Count,” page 5-146.
- Associated option set and group name as well as the conditions/action options specified. See “Creating conditional breakpoints,” page 5-137.
- Last Event Hit shows the breakpoint last encountered.

Integrated debugger features

Add breakpoint

Use the Add Breakpoint dialog box to create a breakpoint. The options that appear in the middle of the dialog box change according to the breakpoint type selected:

- Source
- Address
- Data Watch
- C++ Exception

The following options always display on the right side of the dialog box:

- Other



If you want to set conditions and actions that control breakpoint behavior, click Advanced to open the Breakpoint Condition/Action Options dialog box.


Other

Contains the following options:

- Options** Indicates the name of the option set that defines breakpoint behavior.
- Group** Indicates the name of group to which the breakpoint belongs.

Source breakpoint

Sets a breakpoint on a line in your source code.

- File** Indicates the file that contains the source code where the breakpoint is set.
- Line #** Indicates the line in the source file on which the breakpoint is set.
-  If you select a line of code in an Edit window and choose Add Breakpoint from the SpeedMenu, the debugger completes these settings for you.

Address breakpoint


Sets a breakpoint on a machine instruction.

- Offset** Indicates the address of the machine instruction on which the breakpoint is set.

Data watch breakpoint

Use a Data watch breakpoint to pause your program when a specific location in memory changes value. Data Watch breakpoints (also called *watchpoints* or *changed memory breakpoints*) let you monitor expressions that evaluate to a specific data object or memory location. Data watch breakpoints are monitored continuously during your program's execution.

Because the debugger checks the breakpoint conditions after the execution of every machine instruction, data watch breakpoints are excellent tools for pinpointing code that is corrupting data.

- Address** Enter a specific starting address or any symbol (such as a variable or a class data member or method) that evaluates to an address.
-  If you enter an address expression that evaluates to a memory location that contains executable code, the Data watch breakpoint behaves like an Address breakpoint; the breakpoint fires when the code at the specified address is executed.
- Length** When entering an address expression symbol, you can also enter a count of the number of bytes you want monitored.

For example, coding in C, suppose you have declared the following array:

```
int string[81];
```

You can watch for a change in the first ten elements of this array by entering the following item into the Condition Expression input box:

```
&string[0], 40
```

The area monitored is 40 bytes long which equals ten elements in the array (an **int** is 4 bytes).

C++ exception breakpoint

Sets a breakpoint that pauses your program when it throws or catches a C++ exception.

- Type** Specifies the data type (such as **int**, **long**, **char**, or a class name) used with the exception. If you enter an ellipses (...) into the Type field, the debugger will trap any C++ exception that is thrown or caught by your program.
- Stop on Throw** Pauses program execution when an exception is thrown.
- Stop on Catch** Pauses program execution when an exception is caught.

Stop on Destructor Pauses program execution when any object is destroyed (when a destructor is called) after an exception is thrown.

Breakpoint Condition/Action Options

Use this dialog box to:

- Specify settings that control the behavior of one or more breakpoints, such as the conditions under which a breakpoint is activated and the type of actions that take place when it does.
- Enable and disable breakpoint groups

To display this dialog box, use any of the following methods:

- Choose Debug|Breakpoint Options.
- Choose Debug|Add Breakpoint and click the Advanced button on the Add Breakpoint window.
- Choose View|Breakpoint and double-click a breakpoint listed in the Breakpoints window. Then click the Advanced button on the Edit Breakpoint window.

The Breakpoint Condition/Action Options dialog box contains the following options:

Names	Lists the names of Option sets that have been created.
Conditions	Provides settings that determine when and where a breakpoint is activated.
Actions	Provides settings that determine what actions take place when a breakpoint is activated.

Names Lists the names of existing option sets. Use the checkbox next to each option set to enable or disable it.

For example, if you clear the checkbox next to an option set called `MyOptionSet`, the debugger ignores its settings and all breakpoints that use this option set behave like unconditional breakpoints. To reactivate the breakpoint settings in `MyOptionSet` so that they will be used by the debugger, click its checkbox.

Conditions This group of settings determines when and where a breakpoint is activated:

Expr. True	Each time the debugger encounters the breakpoint, it evaluates an expression to determine if the breakpoint should activate.
Pass Count	Indicates the number of times the debugger encounters the breakpoint line before it activates.



Click Add or Delete to create or remove an option set.

Expr. True Enter the expression you want to evaluate each time the debugger reaches the breakpoint. If the expression becomes true (nonzero) when the breakpoint is encountered, the debugger activates the breakpoint and carries out any actions specified for it. You can enter a Boolean expression that, for instance, tests if a value falls within a certain range or if a flag has been set.

For example:

If you enter the expression

`x == 1`

the debugger activates the breakpoint only if *x* has been assigned the value 1 at the time the breakpoint is encountered.

If you enter the expression

`x > 3`

and select Break, when the debugger reaches the breakpoint, your program pauses if the current value of *x* is greater than 3. Otherwise, the breakpoint is ignored.

Pass Count This option includes the following settings:

- | | |
|---------|---|
| Up to | Specifies the number of times you want the debugger to reach the breakpoint before it is activated. |
| Current | Shows the actual number of times the debugger has reached the breakpoint so far. You can change this setting if your want to. |

Unconditional breakpoint example

Suppose Break is checked, and in the Pass Count box you enter 2. In this case, your program does not stop until the second time the debugger reaches the breakpoint.



Conditional breakpoint example

Suppose Break is checked, plus you enter the expression `x > 3` and in the Pass Count box you enter 2. In this case, your program does not stop until the second time the debugger reaches the breakpoint (that is, when the value of *x* is greater than 3).

Actions This group of options lets you specify the actions you want carried out each time the breakpoint is activated:

- | | |
|---------------|---|
| Break | Pauses program execution |
| Stop Log | Stops posting debugger generated messages |
| Start Log | Starts posting debugger generated messages |
| Log Expr | Displays the value of an expression in the message window |
| Eval Expr | Evaluates an expression |
| Log Message | Displays a message in the message window |
| Enable Group | Reactivates a group of breakpoints |
| Disable Group | Disables a group of breakpoints |

Break Click Break (the default) to pause program execution when the debugger activates the breakpoint. Clear this checkbox if you do not want your program to pause at the breakpoint.

Stop Log	Stops displaying debugger messages in the Run-time Tab of the Message window when the breakpoint is activated.
Start Log	Starts displaying debugger messages in the Run-time Tab of the Message window when the breakpoint is activated.
Log Expr	<p>Click Log Expr if you want to display the value of an expression in the Run-time tab of the Message window. Then, enter the expression in the input box next to it. The debugger logs the value each time the breakpoint activates. Use this option when you want to output a value each time you reach a specific place in your program — this technique is known as instrumentation.</p> <p>For example, you can place a breakpoint at the beginning of a routine and set it to log the values of the routine arguments. Then, after running the program, you can determine where the routine was called from, and if it was called with erroneous arguments. This will give you no idea where it was called from, but will tell you what the arguments are.</p> <p> When you log expressions, be careful of expressions that unexpectedly change the values of variables or data objects (side effects).</p>
Eval Expr	Click Eval Expr if you want the breakpoint to evaluate an expression. Then, enter an expression in the input box next to it. For best results, use an expression that changes the value of a variable or data object (side effects).
<i>You cannot use this technique to directly modify your compiled program.</i>	By “splicing in” a piece of code before a given source line, you can effectively test a simple bug fix; you do not have to go through the trouble of compiling and linking your program just to test a minor change to a routine.
Log Message	Click Log message if you want the breakpoint to display a message in the Run-time tab of the Message window when the breakpoint is activated. Then, enter the text of the message in the input box next to it.
Disable Group	Click Disable group if you want the breakpoint to disable a group of breakpoints. Then, enter a group name in the input box next to it.
	When a group of breakpoints is disabled, the breakpoints are not erased, they are simply hidden from the debugger until you enable them.
Enable Group	Click Enable group if you want the breakpoint to reactivate a group of breakpoints that have been previously disabled. Then, enter a group name in the input box next to it.
Add Conditions/Actions	Enter a name for the option set and click OK to create a new set of breakpoint options. Then enter your selections using the Breakpoint Condition/Action options dialog box.

Edit Breakpoint dialog box

Use this dialog box to modify an existing breakpoint. The options that appear on left side of the dialog box change according to the breakpoint type selected.

The integrated debugger provides the following types of breakpoints:

- Source
- Address
- Data watch
- C++ Exception

The following options always display on the right side of the dialog box:

- Other



If you want to set conditions and actions that control breakpoint behavior, click Advanced to open the Breakpoint Condition/Action options dialog box.

Examining program data values

Even though you can discover many interesting things about your program by running and stepping through it, you'll usually need to examine the values of program variables to uncover bugs. For example, it's helpful to know the value of the index variable as you step through a **for** loop, or the values of the parameters passed to a function call.

After you have paused your application within the integrated debugger, you can examine the different symbols and data structures with regards to the location of the current *execution point*.

You can view the state of your program by:

- Watching program values
- Inspecting data elements.
- Evaluating expressions
- Viewing the low-level state of your program
- Viewing functions in the Call Stack window

You can also use the Browser to view the global variables and classes contained in your program.

Modifying program data values

Sometimes you will find that a programming error is caused by an incorrect data value. Using the integrated debugger, you can test a "fix" by modifying the data value while your program is running. You can modify program data values using:

- The Evaluate dialog box.
- The Inspector window's Change SpeedMenu command
- A breakpoint's Evaluate action, set from the Breakpoint Condition/Action dialog box
- The CPU window's Dump pane
- The Register & Stack window

Understanding watch expressions

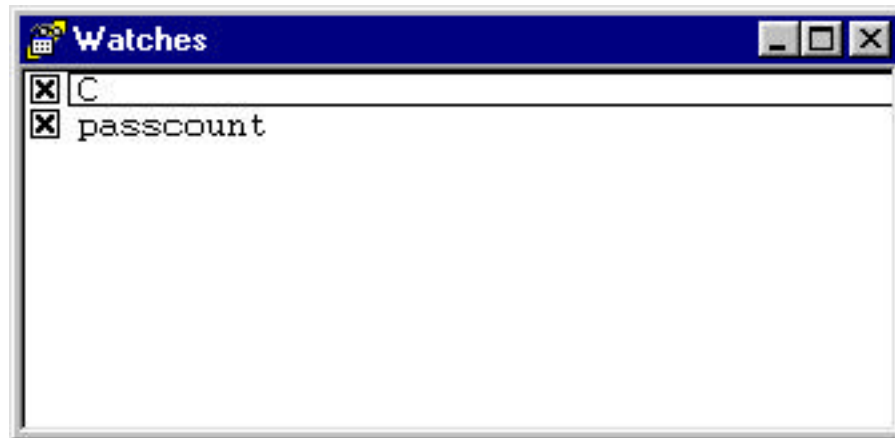
You use *watches* to monitor the changing values of a variables or expressions during your program run. After you enter a watch expression, the Watches window displays the

current value of the expression based on the scope of the *execution point*. Each time your program pauses (such as when it encounters a breakpoint), the value of the watch changes to reflect the current value of the expression according to the values of the variables in your program.

Using Watches window

To display the Watches window, choose View|Watch.

Figure 5-4
Watches window



The Watches window lists the watches you are currently monitoring. Check the checkbox beside a watch to enable it. Clear the checkbox beside a watch to disable it.

The Watches window will be blank if you have not added any watches.

The left side of the Watches window lists the expressions you enter as watches and their corresponding data types and values appear on the right. The values of compound data objects (such as arrays and structures) appear between braces ({ }).



If the execution point steps out of the scope of a watch expression, the watch expression is undefined. When the execution point re-enters the scope of the expression, the Watches window again displays the current value of the expression.

Adding a watch

You can add a watch the following ways:

- Place the insertion point on a word in an Edit window and choose Watch from the Edit window SpeedMenu. The debugger adds a watch on the expression at the insertion point and opens the Watches window.
- From the Watches window, right-click to bring up the Watches window SpeedMenu and choose Add Watch. In the Add Watch dialog box, create a watch expression on any variable or expression available to the program you are debugging.
- Bring up the Add Watch dialog box by choosing Debug|Add Watch and enter a variable or expression you would like to watch.

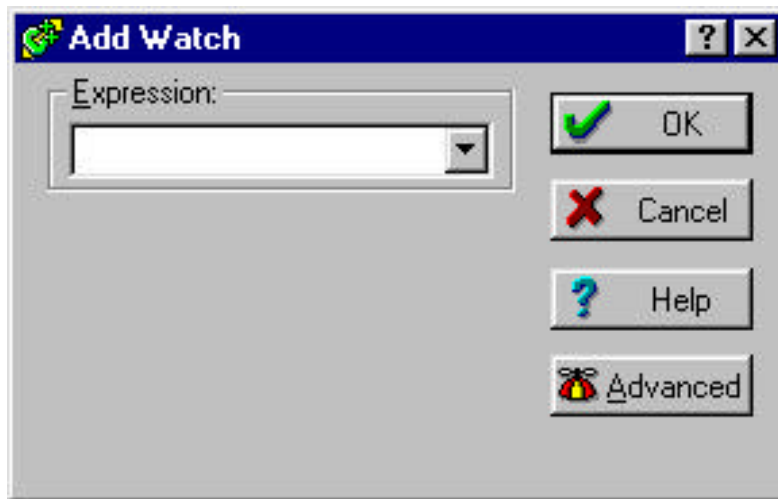
Add Watch dialog box

The Add Watch dialog box lets you monitor the value of both simple variables (such as integers) and compound data objects (such as arrays). In addition, you can watch the values of calculated expressions that do not refer directly to memory locations. For example, you could watch the expression $x * y + 4$.

To create a watch expression using the Add Watch dialog box:

1. Choose Debug|Add Watch or choose Add Watch from the Watches window SpeedMenu.

Figure 5-5
Add Watch
dialog box



2. Enter an *expression* into the Expression input box.
3. Click OK to add the watch or choose any of the following optional settings:
 - Advanced

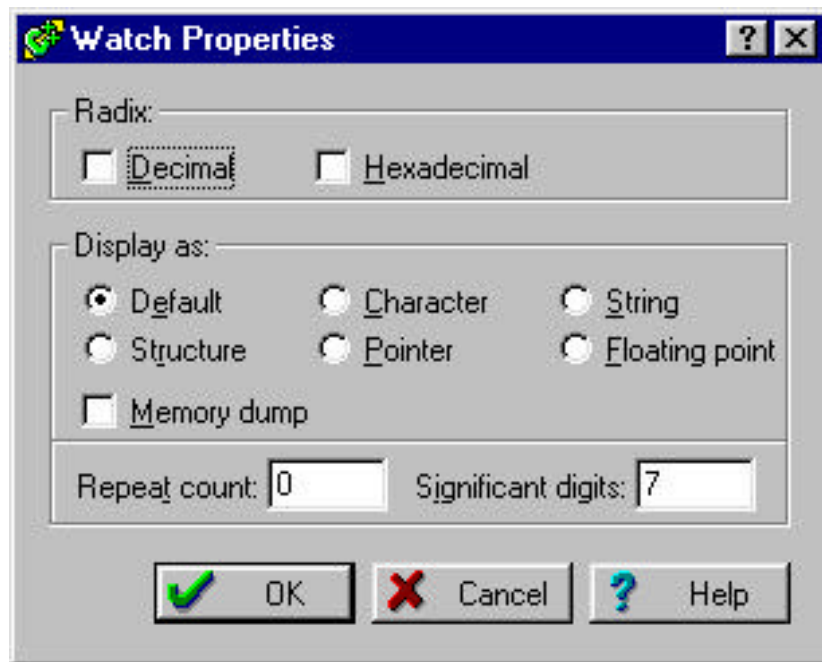


After you add the watch expression, the Paradigm C++ IDE automatically opens the Watches window if it is not already open.

Formatting watch expressions

You can format the display of a watch expression using the Watch Properties dialog box. Click Advanced from the Add Watch dialog box to bring up the Watch Properties dialog box.

Figure 5-6
Watch Properties
dialog box



By default, the debugger displays integer values in decimal form. However, by checking the Hexadecimal button in the Watch Properties dialog box, you can specify that an integer watch be displayed as hexadecimal. You can also vary the display of the watches using the Display As buttons in the Watch Properties dialog box.



For more on Display As buttons in the Watch Properties dialog box, select the Display As button you would like help on and hit *F1* for online Help.

To format a floating-point expression, click the Floating Point button, then indicate the number of significant digits you want displayed in the Watch window by typing this number in the Significant Digits text box.

If you're setting up a watch on an element in a data structure such as an array), you can display the values of consecutive data elements. For example, suppose you have an array of five integers named *xarray*. Type the number 5 in the Repeat Count text box of the Watch Properties dialog box to see all five values of the array.

You can also format watch expressions using the expression format specifiers shown in Table 5-1, page 5-155. Format specifier settings override any settings specified in the Watch Properties dialog box. Format specifiers use the following syntax:

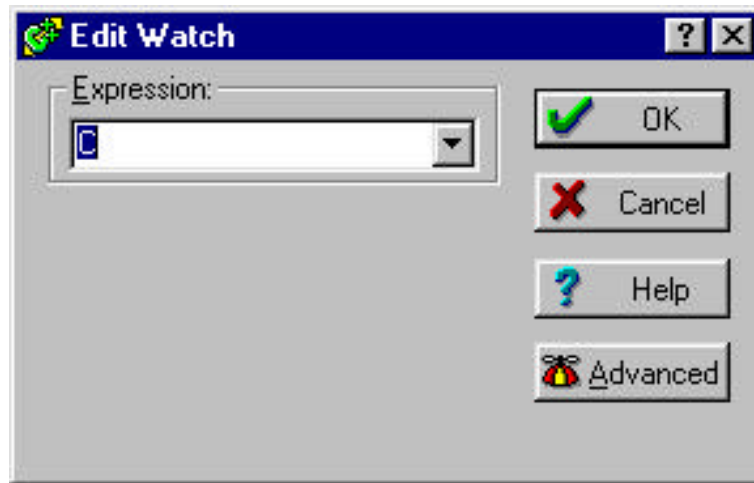
```
expression [, format_specifier]
```

Changing watch properties

To change the properties of a watch:

1. Choose View|Watch, to open the Watches window.
2. Double-click a watch to open the Edit Watch dialog box.

Figure 5-7
Edit Watch dialog
box



Edit Watch dialog box

Use this dialog box to change the settings for a watch expression:

1. Either accept or change the information in either of the following options:
2. Either
 - Choose OK to save your changes and close the dialog box.
 - Click Advanced to open the Watch Properties if you want to change how a watch expression displays in the Watches window.

Disabling and enabling watches

Evaluating many watch expressions can slow down the process of debugging. Disable a watch expression when you prefer not to view it in the Watches window, but want to save it for later use.

To enable or disable a watch

1. Choose View|Watch to open the Watch window.
2. Either
 - Click the checkbox next to a watch to enable it.
 - Clear the checkbox next to a watch to disable it.

To disable or enable selected watches

1. Hold down the *Shift* or *Ctrl* key and click on one or more watches in the Watch window.
2. Choose Enable or Disable watches from the Watch window SpeedMenu.

Deleting a watch

You can delete a watch the following ways:

1. Choose View|Watch to display the Watches window.
2. Select one or more watch expressions. (To make multiple selections, hold down the *Shift* or *Ctrl* key and click.)
3. Choose Remove Watch(es) on the SpeedMenu.

Dynamic updates

The Dynamic update dialog box controls the behavior of memory reads and peripheral register reads while running. Inspector and Watch windows can be updated dynamically while running, if the option Allow memory reads while running is enabled. Peripheral register viewers can also be dynamically updated while running if the option Allow peripheral reads while running is enabled. Enabling these options will interrupt target execution. Do not enable these options if you wish non-intrusive execution of your application.



These options only apply to remote debugging solutions with the ability to interrupt target execution.

Inspecting data elements

You can use inspect windows to examine and modify data values. Inspect windows are extremely useful because they format the data according to the type of data being viewed; there are different types of Inspect windows for scalars, arrays, structures, functions, and classes with and without member functions.

The easiest way to inspect a data item is to highlight the expression you want to inspect (or just position the text cursor on the token) in the Edit window, and choose Inspect Object from the SpeedMenu (or press *Alt-F5*). If you inspect expressions using this method, the expression is always evaluated within the scope of the line on which the expression appears.

You can also inspect data expressions using the following method,

1. Choose Debug|Inspect to display the Inspect Expression window.
2. Type the expression you want to inspect, then choose a previously entered expression from the drop down list.
3. Choose OK to display an Inspector window.

If the execution point is in the scope of the expression you are inspecting, the value appears in the Inspect window. If the execution point is outside the scope of the expression, the value is undefined.

If you are inspecting a compound data item, such as an array or a structure, you can view the details of the data item by opening another Inspect window on the element you want to inspect.

To inspect an element of a compound data item:

1. In the Inspector window, select the item you want to inspect.
2. Choose Inspect on the Inspector window SpeedMenu, or press *Enter*.

You can also use Inspector windows to change the value of a single data item:

1. Select the data item whose value you want to modify.
2. Choose Change on the Inspect window SpeedMenu.
3. Type the new value into the Change value dialog box and click OK.

If you are inspecting a data structure, it is possible the number of items displayed might be so great that you will have to scroll in the Inspector window to see the data you want. For easier viewing,

Narrow the display to a range of data items:

1. Left-click in the Inspect window or choose Set Range from the SpeedMenu.
2. In the Starting Index text box, enter the index of the first item you want to view.
3. In the Count text box, enter the number of items you want to see in the Inspect window.

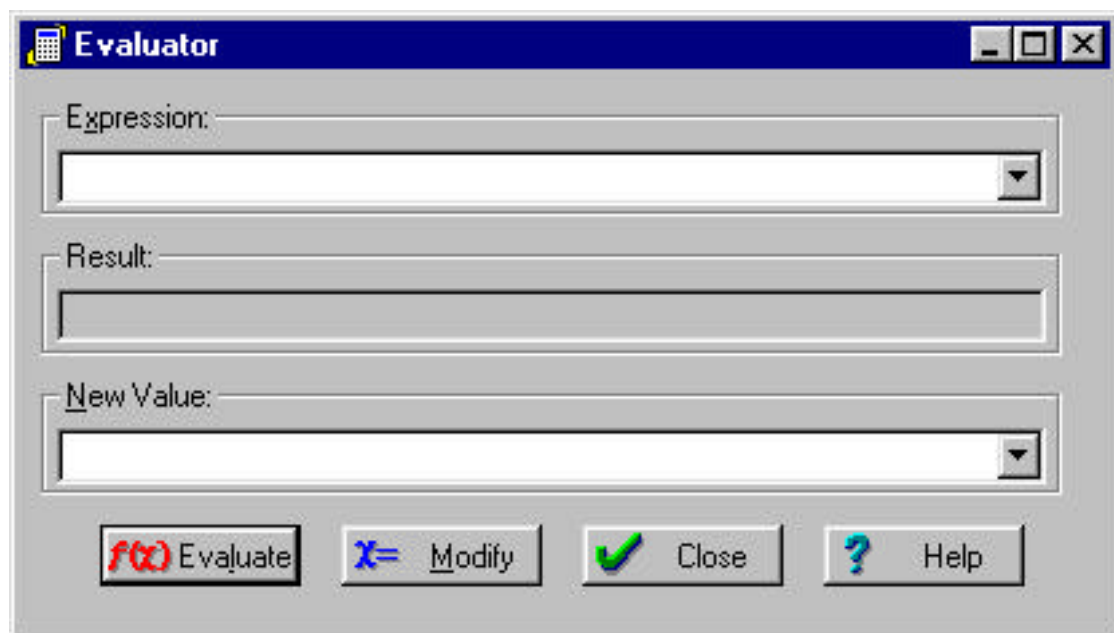
Evaluating and modifying expressions

You can evaluate expressions using the Expression Evaluator dialog box. The Expression Evaluator dialog box has the advantage that it lets you change the values of variables and items in data structures during the course of your debugging session. This can be useful if you think you've found the solution to a bug, and you want to try it out before exiting the debugger, changing the source code, and recompiling the program.

Evaluating expressions

Choose Debug|Evaluate to open the Expression Evaluator dialog box. By default, the token at the cursor position in the current Edit window is placed in the Expression text box. You can accept or modify this expression, enter another one, or choose an expression from the history list of expressions you've previously evaluated.

Figure 5-8
Evaluator dialog
box



To evaluate the expression, click the Evaluate button. Using this dialog box, you can evaluate any valid language expression, except ones that contain:

- Local or static variables that are not accessible from the current execution point
- Symbols or macros defined with **#define**

When you evaluate an expression, the current value of the expression is displayed in the Result field of the dialog box. If you need to, you can format the result by adding a comma and one or more format specifiers to the end of the expression entered in the Expression text box. Table 5.1, page 5.155 details the legal format specifiers.

Table 5-1
Expression
format specifiers

Character	Types affected	Function
H or X	Integers	Hexadecimal. Shows integer values in hexadecimal with the 0x prefix, including those in data structures.
C	Char, strings	Character. Shows special display characters for ASCII 0-31. By default, such characters are shown using the appropriate C escape sequences (/n, /t, and so on).
D	Integers	Decimal. Shows integer values in decimal form, including those in data structures.
F <i>n</i>	Floating point	Floating point. Shows <i>n</i> significant digits (where <i>n</i> is in the range of 2-18, and 7 is the default).
nM	All	Memory dump. Shows <i>n</i> bytes starting at the address of the indicated expression. If <i>n</i> is not specified, it defaults to the size in bytes of the type of the variable. By default, each byte displays as two hex digits. The C, D, H, S, and X specifiers can be used with M to change the byte formatting.
P	Pointers	Pointer. Shows pointers in <i>seg:ofs</i> instead of the default <i>Ptr(seg:ofs)</i> . It tells you the region of memory in which the segment is located and, if appropriate, the name of the variable at the offset address.
R	Structures, unions	Structure/Union. Shows field names and values such as (X:1;Y:10;Z:5) instead of (1,10,5).
S	Char, strings	String. Shows ASCII 0-31 as C escape sequences. Use only to modify memory dumps (see <i>nM</i> above).

For example, to display a result in hexadecimal, type ,H after the expression. To see a floating-point number to 3 decimal places, type ,F 3 after the expression.

You can also use a *repeat count* to reference a specific number of data items in arrays and structures. To specify a repeat count, follow the expression with a comma and the number of data items you want to reference. For example, suppose you declared the following array in your program:

```
int my_array[10] ;
```

The following expression evaluates the first 5 elements of this array and displays the result in hexadecimal:

```
my_array, 5h
```

Modifying the values of variables

Once you've evaluated a variable or data structure item, you can modify its value. Modifying the value of data items during a debugging session lets you test different bug hypotheses and see how a section of code behaves under different circumstances.

To modify the value of a data item:

1. Open the Expression Evaluator dialog box and enter the name of the variable you want to modify into the Expression input box.
2. Click Evaluate to evaluate the data item.
3. Type a value into the New Value text box (or choose a value from the drop down list), then click Modify to update the data item.



When you modify the value of a data item through the debugger, the modification is effective for that specific program run only; the changes you make through the Expression Evaluator dialog box do not affect your program source code or the compiled program. To make your change permanent, you must modify your program source code in the Edit window, then recompile your program.

Keep these points in mind when you modify program data values:

You can change individual variables or elements of arrays and data structures, but you cannot change the entire contents of an array or data structure.

- The expression in the New Value text box must evaluate to a result that is assignment-compatible with the variable to which you want to assign it. A good guideline is if that assignment would cause a compile-time or run-time error, it is not a legal modification value.

Warning! Modifying values (especially pointer values and array indexes), can have undesirable effects because you can overwrite other variables and data structures. Use caution whenever you modify program values from the debugger.

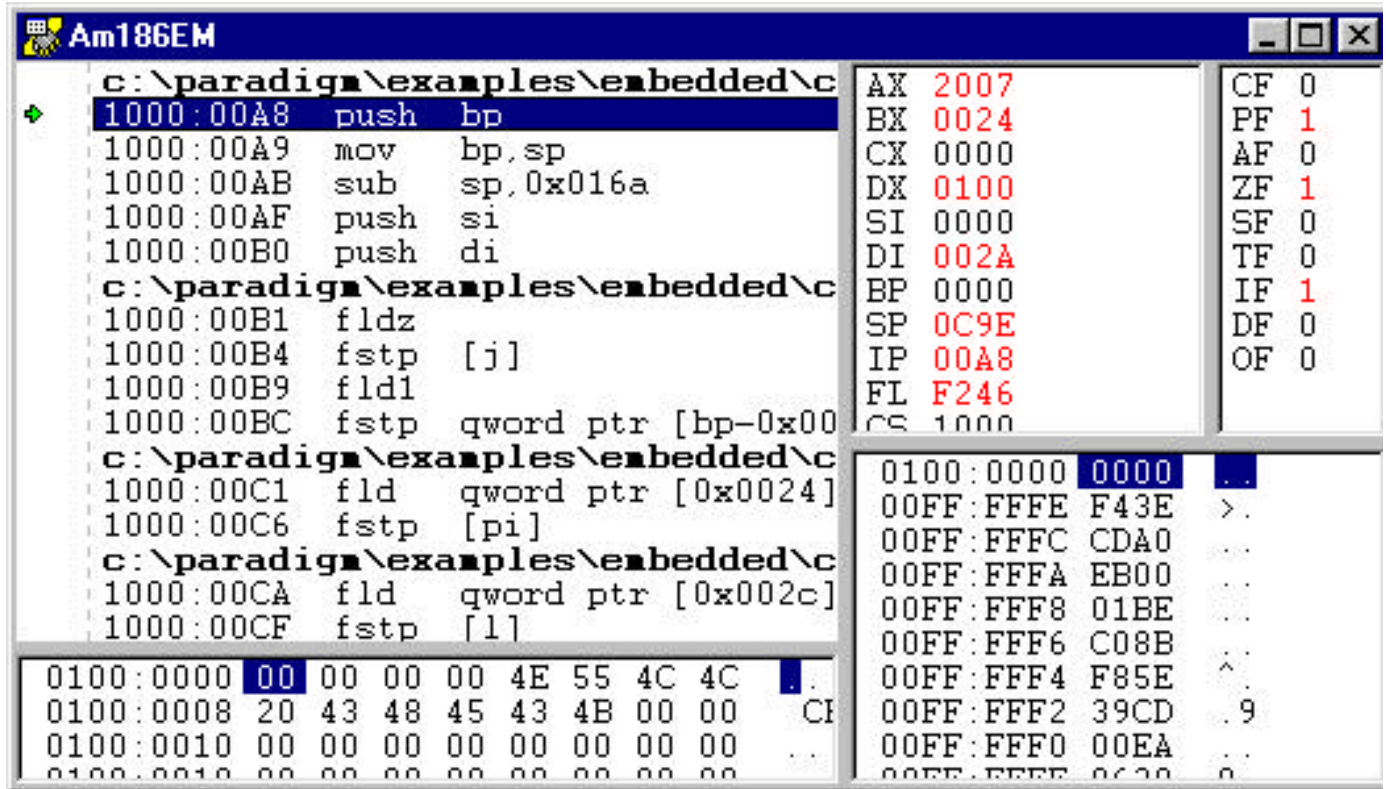
CPU window

The CPU window consists of five separate panes. Each pane gives you a view into a specific low-level aspect of your running application:

- The **Disassembly pane** displays the assembly instructions that have been disassembled from your application's machine code. In addition, the Disassembly pane displays the original program source code above the associated assembly instructions.
- The **Dump pane** displays a memory dump of any memory accessible to the currently loaded executable module. By default, memory is displayed as hexadecimal bytes.
- The **Stack Pane** displays the current contents of the program stack. By default, the stack is displayed as hexadecimal bytes.
- The **Registers pane** displays the current values of the CPU registers.
- The **Flags pane** displays the current values of the CPU flags.

Each pane has an individual SpeedMenu that provides commands specific to the contents of that pane and the target processor.

Figure 5-9 CPU window



Resizing the CPU window panes

You can customize the layout of the CPU window by resizing the panes within the window. Drag the pane borders within the window to enlarge or shrink the windows to your liking.

The Disassembly pane

The left side of the Disassembly pane lists the address of each disassembled instruction. An arrow to the right of the memory address indicates the location of the current execution point. To the right of the memory addresses, the Disassembly pane displays the assembly instructions that have been disassembled from the machine code produced by the compiler. If you are viewing code that has been linked with a symbol table, the debugger displays the source code that is associated with the disassembled instructions.

The Disassembly pane SpeedMenu

The Disassembly pane supports the following keyboard commands:

- Press *Ctrl-N* to set the instruction pointer (the value of IP/EIP register) to the beginning of the statement that you have highlighted in the Disassembly pane. Note that this is not the same as stepping through the instructions; the debugger does not execute any instructions that you might skip.
- Press *Ctrl+LeftArrow* and *Ctrl+RightArrow* to shift the starting point of the display up or down one byte. Beware that changing the starting point of the display in the Disassembly pane changes where the debugger begins disassembling the machine code.



The debugger displays dashes if you view a program memory location in which nothing is loaded.

The Disassembly pane has the following SpeedMenu commands:

- Run To Current
- Set PC To Current
- Toggle Breakpoint
- Go to Address
- Go to Current PC
- Follow jump <address> into Disassembly pane
- Follow address <address> into Memory Dump pane
- Show previous address
- Go to source

Run to Current

The Run To Current command lets you run your program at full speed to the instruction that you have selected in the Disassembly pane. After your program is paused, you can use this command to resume debugging at a specific program instruction.

Set PC to current

The Set PC to Current command changes the location of the program counter (the value held in the IP/EIP register) to the currently highlighted line in the Disassembly pane. When you resume program execution in the debugger, it starts at the new address. This command is useful when you want to skip certain machine instructions.



Use this command with extreme care; it is easy to place your system in an unstable state when you skip over program instructions.

Toggle Breakpoint

When you choose Toggle Breakpoint, the debugger sets an unconditional or "simple" breakpoint at the instruction which you have selected in the Disassembly pane. A simple breakpoint has no conditions, and the only action is that it will pause the program's execution.

If a simple breakpoint exists on the selected instruction, then Toggle Breakpoint will delete the breakpoint at that code location.

Go to Address

The Go to Address command prompts you for a new area of memory to display in the Code, Dump, or Machine Stack panes of the CPU window. Enter any expression that evaluates to a memory location, such as **main()**. Be sure to precede hexadecimal values with 0x.



The debugger displays dashes if you try to access an address that is not within the scope of the application you are debugging.

You can also press *Ctrl+LeftArrow* and *Ctrl+RightArrow* to shift the starting point of the display up or down one byte.

Go to current PC

This command positions the Disassembly pane at the location of the current program counter (the location indicated by the IP/EIP register). This location indicates the next instruction to be executed by your program.

This command is useful when you have navigated through the Disassembly pane, and you want to return to the next instruction to be executed.

**Follow
jump into
Disassembly
pane**

This command highlights in the Disassembly pane the destination address of the currently highlighted instruction. Use this command in conjunction with instructions that cause a transfer of control (such as **CALL**, **JMP**, and **INT**) and with conditional jump instructions (such as **JZ**, **JNE**, **LOOP**, and so forth). For conditional jumps, the address is shown as if the jump condition is **TRUE**. Use the **Show Previous Address SpeedMenu** command to return to the origin of the jump.

From the Memory Dump pane, set the display to Longs for best results.

**Follow
address into
Dump pane**

This command highlights in the Memory Dump pane the address of the currently highlighted address. The **Show Previous Address SpeedMenu** command returns you to the address from where you jumped.

**Show
previous
address**

This command restores the CPU window to the display it had before you issued the last **Follow Address** command. The **Follow Address** commands are found on the **SpeedMenus** of the Disassembly pane, the Machine Stack pane, and the Memory Dump pane of the CPU window.

Go to source

The **Go to source** command activates the Edit window and positions the insertion point at the source code that corresponds to the disassembled instruction selected in the Disassembly pane. If there is no corresponding source code (for example, if you're examining Windows kernel code), this command has no effect.

Memory Dump pane

The Dump pane displays the raw values contained in addressable areas of your program. The display is broken down into three sections: the memory addresses, the current values in memory, and an ASCII representation of the values in memory.

By default, the Dump pane displays the memory values in hexadecimal notation. The leftmost part of each line shows the starting address of the line. Following the address listing is an 8-byte hexadecimal listing of the values contained at that location in memory. Each byte in memory is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory. Non-printable values are represented with a period.

The format of the memory display depends on the format selected with the **Display As SpeedMenu** command. If you choose one of the floating-point display formats (**Floats** or **Doubles**), a single floating-point number is displayed on each line. The **Bytes** format displays 8 bytes per line, **Words** displays 4 words per line, and **Longs** displays 2 long words per line.



You can press **Ctrl+LeftArrow** and **Ctrl+RightArrow** to shift the starting point of the display up or down one byte. Using these keystrokes is often faster than using the **Go to Address** command to make small adjustments to the display.

The Dump pane SpeedMenu

The Dump pane has the following SpeedMenu commands:

- Go to Address
- Display As
- Follow address <address> into Disassembly pane
- Follow address <address> into Memory Dump pane
- Follow address <address> into Machine Stack pane
- Show previous address



You can change the values of memory displayed in the Dump pane by pressing the *Ins* key and typing into the display (when you press *Ins*, the insertion point in the pane shrinks to highlight a single nibble in memory). Be extremely careful when changing program memory values; even small changes in program values can have disastrous effects on your running program.

Display as Use the Display As command to format the data that's listed in the Dump or Stack pane of the CPU window. You can choose any of the data formats listed in the following table:

Table 5-2
Data formats

Data type	Display format
Bytes	Displays data in hexadecimal bytes
Words	Displays data in 2-byte hexadecimal numbers
Longs	Displays data in 4-byte hexadecimal numbers
Floats	Displays data in 4-byte floating-point numbers using scientific notation
Doubles	Displays data in 8-byte floating-point numbers using scientific notation

Follow address into Disassembly pane This command highlights in the Disassembly pane the address of the currently highlighted address. The Show Previous Address SpeedMenu command returns you to the address from where you jumped.

From the Memory Dump pane, set the display to Longs for best results.

Follow address into Stack pane This command highlights in the Machine Stack pane the address of the currently highlighted address. The Show Previous Address SpeedMenu command returns you to the address from where you jumped.

Set the display to Longs for best results.

Machine Stack pane

The Stack pane displays the raw values contained in the your program stack. The display is broken down into three sections: the memory addresses, the current values on the stack, and an ASCII representation of the stack values.

The debugger displays dashes if you view an unloaded program memory location.

By default, the Machine Stack pane displays the memory values in hexadecimal notation. The leftmost part of each line shows the starting address of the line. Following the address listing is a 4-byte listing of the values contained at that memory location. Each byte is represented by two hexadecimal digits. Following the hexadecimal display is an ASCII display of the memory; non-printable values are represented with a period.

The format of the memory display depends on the format selected with the Display As SpeedMenu command. If you choose one of the floating-point display formats (Floats or Doubles), a single floating-point number is displayed on each line. The Bytes format displays 4 bytes per line, Words displays 2 words per line, and Longs (the default) displays 1 long word per line.



You can press *Ctrl+LeftArrow* and *Ctrl+RightArrow* to shift the starting point of the display up or down one byte. Using these keystrokes is often faster than using the Go to Address command to make small adjustments to the display.

The Stack pane SpeedMenu

The Stack pane has the following SpeedMenu commands:

- Go to Address
- Go to Top Frame
- Go to Top of Stack
- Display As
- Follow address <address> into Disassembly pane
- Follow address <address> into Memory Dump pane
- Follow address <address> into Machine Stack pane
- Show previous address



You can change the values of memory displayed in the Stack pane by pressing the *Ins* key and typing into the display (when you press *Ins*, the insertion point in the pane shrinks to highlight a single nibble in memory). Be extremely careful when changing program memory values; even small changes in program values can have disastrous effects on your running program.

Go to top frame

Positions the insertion point in the Stack pane at the address of the frame pointer (the address held in the BP/EBP register).

Go to top of stack

Positions the insertion point in the Stack pane at the address of the stack pointer (the address held in the SP/ESP register).

Registers pane

The Registers pane displays the contents of the CPU registers of the processor. These registers consist of eight 16-bit 32-bit general purpose registers, six 16-bit segment registers, and the program counter (IP/EIP), and the flags register (FL/EFL).


After you execute an instruction, the Registers pane highlights in red any registers that have changed value since the program was last paused.

The Registers pane SpeedMenu

The Registers pane has the following SpeedMenu commands:

- Increment Register
- Decrement Register
- Zero Register
- Change Register

- Show Old Registers/Show Current Registers'

Increment register	Increment Register adds 1 to the value in the currently highlighted register. This lets you test “off-by-one” bugs by making small adjustments to the register values.
Decrement register	Decrement Register subtracts 1 from the value in the currently highlighted register. This lets you test “off-by-one” bugs by making small adjustments to the register values.
Zero register	The Zero Register command sets the value of the currently highlighted register to 0.
Change register	Lets you change the value of the currently highlighted register. This command opens the Change Register dialog box where you enter a new value. You can make full use of the expression evaluator to enter new values. Be sure to precede hexadecimal values with 0x.
Show old registers	This command toggles between Show old Registers and Show current registers. When you select Show old registers, the Registers pane displays the values which the registers had before the execution of the last instruction. The menu command then changes to Show current registers, which changes the display back to the current register values.
	You can change the values of memory displayed in the Registers pane by pressing the <i>Ins</i> key and typing into the display (when you press <i>Ins</i> , the insertion point in the pane shrinks to highlight a single nibble in memory). Be extremely careful when changing register values; even small changes can have disastrous effects on your running program.

Flags pane

The Flags pane shows the current state of the flags and information bits contained in the processor flags register. After you execute an instruction, the Flags pane highlights in red any flags that have changed value since the program was last paused.

The processor uses the following bits in this register to control certain operations and indicate the state of the processor after it executes certain instructions:

Table 5-3
Flags pane
indicators

Letters in pane	Flag/bit name	EFL register bit number
CF	Carry flag	0
PF	Parity flag	2
AF	Auxiliary carry	4
ZF	Zero flag	6
SF	Sign flag	7
TF	Trap flag	8
IF	Interrupt flag	9
DF	Direction flag	10
OF	Overflow flag	11
IO	I/O privilege level	12 and 13
NF	Nested task flag	14
RF	Resume flag	16

VM	Virtual mode	17
AC	Alignment check	18

The Flags pane SpeedMenu

The Flag pane has the following SpeedMenu commands:

- Toggle Flag



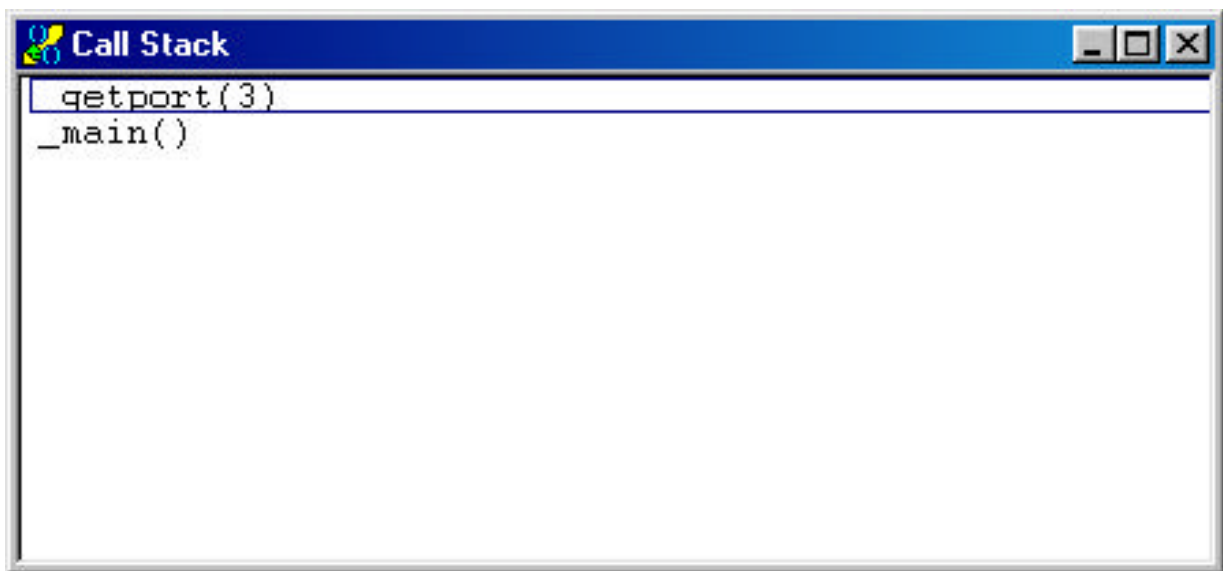
You can change the values of memory displayed in the Flags pane by pressing the *Ins* key and typing into the display (when you press *Ins*, the insertion point in the pane shrinks to highlight a single binary value in memory).

Toggle flag The flag and information bits in the Flags pane can each hold a binary value of 0 or 1. This command toggles the selected flag or bit between these two binary values.

Viewing function calls

While debugging, it can be useful to know the order of function calls that brought you to your current program location. Using the Call Stack window, you can view the current sequence of function calls. The Call Stack window is also helpful when you want to view the arguments passed to a function call; each function listing in the window is followed by a listing that details the arguments with which the call was made. Use View|Call Stack to display the Call Stack window.

Figure 5-10 Call Stack window



In the Call Stack window, the function that is currently executing is listed on top, with all previously called functions listed in sequence below. The bottom of the list always shows the first function in the calling sequence.

The call stack is particularly useful if you accidentally step through code that you wanted to step over. Using the Call Stack window, you can return to the point where the current function was called from, and then resume debugging from there:

1. In the Call Stack window, double-click the function that called the function you accidentally stepped into (it will be the second function listed in the Call Stack window). The Edit window becomes active with the cursor positioned at the location of the function call.
2. In the Edit window, move the cursor to the statement following the function call.
3. Choose Run to Cursor on the Edit window SpeedMenu (or press *F4*).

Navigating to function calls

Using the Call Stack window, you can view or edit the source code located at a particular function call. Right-clicking a function in the Call Stack window displays the SpeedMenu, from where you can choose either View Source or Edit Source. Each of these commands causes the Edit window to display the selected function; however, Edit Source gives focus to the Edit window so you can modify the source code at that function location.

If you select the top function in the Call Stack window, these commands cause the Edit window to display the location of the execution point in the current function. Selecting any other function call causes the debugger to display the actual function call in the Edit window.

Paradigm C++ compiler

If you prefer to develop your applications outside of the Paradigm C++ IDE, you can compile and link your programs from the command line using the Paradigm command-line tools. When you develop applications using this method, you must first write your program source code using a text editor, then compile the code into an object (.OBJ) file using the appropriate command-line compiler. After the .OBJ file is generated, you must link all the necessary files to create the final executable program.

Using the command-line compiler

Paradigm C++ includes the following compiler:

- PCC.EXE is the 16-bit compiler.

Command-line compiler syntax

The general syntax for the Paradigm C++ command-line compiler is:

```
PCC [option [option...]] filename [filename...]
```



Items enclosed in brackets are optional. The *option* items refer to the command-line options and *filename* refers to the source-code files you want to compile. A complete summary of command-line options can be found under "command-line options" in the online Help index. A list of command-line options is also on page 3-112.

To see a list of the commonly used compiler options, type PCC at the command line (without any options or file names), then press *Enter*. This list displays the options that are enabled by default.

The command-line compiler name and each option and file name must be separated by at least one space. Precede each option by either a hyphen (-) or a forward slash (/); for example:

```
PCC -Ic:\code\hfiles
```

Options and file names entered on the command line override settings in configuration files.

You can use PCC to send files to PLINK (.OBJ files) or PASM (.ASM files if you have PASM installed on your machine).

Default settings

PCC.EXE has options that are on by default. To turn off a default option or to override options in a configuration file, follow the option with a minus (-) sign.

Files without extensions and files with the .CPP extension compile as C++ files. Files with a .C extension or with extensions other than .CPP, .OBJ, .LIB, or .ASM compile as C files.

Compiler configuration files

If you repeatedly use a certain set of options, you can list them in a configuration file instead of continually typing them on the command line. A configuration file is a standard ASCII text file that contains one or more command-line options. Each option must be separated by a space or a new line.

Whenever you issue a compile command, PCC.EXE searches for a configuration file called PCC.CFG. The compilers look for the .CFG files first in the directory where you issue the compile command, then in the directory where the compilers are located.

You can create and use multiple configuration files in addition to using the default .CFG file. To use a configuration file, use the following syntax where you would place the compiler options:

```
+ [path]filename
```

For example, you could use the following command line to use a configuration file called MYCONFIG.CFG:

```
PCC +C:\MYPROJ\MYCONFIG.CFG mycode.cpp
```

Options typed on the command line override settings stored in configuration files.

Compiler response files

Response files let you list both compiler options and file names in a single file (unlike configuration files, which accept only compiler options). A response file is a standard ASCII text file that contains one or more command-line options and/or file names, with each entry in the file separated by a space or a new line. In addition to simplifying your compile commands, response files let you issue a longer command line than most operating systems allow.

The syntax for using a single response file is:

```
PCC @[path]respfile.txt
```

The syntax for using multiple response files is:

```
PCC @[path]respfile.txt @[path]otheresp.txt
```

Response files shipped with Paradigm C++ have an .RSP extension.

Options typed at the command line override any option or file name in a response file.

Compiler-option precedence rules

The command-line compilers evaluate options from left to right, and follows these rules:

- If you duplicate any option except **-D**, **-U**, **-I**, or **-L**, the last option typed overrides any earlier one. (**-D**, **-U**, **-I**, and **-L** are cumulative.)
- Options typed at the command line override configuration and response file options.

Entering directories for command-line options

Paradigm C++ can search multiple directories for include and library files. This means that the syntax for the library directories (**-L**) and include directories (**-I**) command-line options, like that of the #define option (**-D**), allows multiple listings of a given option. Here is the syntax for these options:

```
Ldirname[;dirname;...]  
Idirname[;dirname;...]
```

The parameter `dirname` used with **-L** and **-I** can be any directory or directory path. You can enter these multiple directories on the command line in the following ways:

- You can stack multiple entries with a single **-L** or **-I** option by using a semicolon:

```
PCC.EXE -Ldirname1;dirname2;dirname3 -Iincl;inc2;inc3 myfile.c
```

- You can place more than one of each option on the command line, like this:

```
PCC.EXE -Ldirname1;dirname2;dirname3 -Iincl;inc2;inc3 myfile.c
```

- You can mix listings:

```
PCC.EXE -Ldirname1;dirname2;dirname3 -Iincl;inc2 -Iinc3  
myfile.c
```

If you list multiple **-L** or **-I** options on the command line, the result is cumulative: The compiler searches all the directories listed, in order from left to right.

*Paradigm C++
also supports
multiple library
directories.*

PLINK uses a configuration file called `PLINK.CFG`, a response file (optional), and command-line options to link object modules, libraries, and resources into an executable `.AXE` file.

Using PLINK

PLINK is a command-line tool that combines object modules (`.OBJ` files), library modules (`.LIB` files), and resources to produce executable files. Because the compiler automatically calls PLINK, you don't need to use PLINK unless you suppress the linking stage of compiling (see the `-c` compiler option).

PLINK is invoked from the command line to link a configuration file called `PLINK.CFG`, an optional response file, and command-line options to link object modules, libraries, and resources into an executable file.

PLINK command-line syntax

The linker syntax controls how the linkers work. Linker command-line options are case-sensitive.

The linker can also use a configuration file called `PLINK.CFG` for options that you'd normally type at the command-line.

Syntax

```
PLINK [@respfile][options] startup myobjs, [exe], [mapfile],  
[libraries]
```

Where items enclosed in brackets are optional.

<code>[@respfile]</code>	A response file is an ASCII file that lists linker options and file names that you would normally type at the command line. By placing options and file names in a response file, you can save the amount of keystrokes you need to type to link your application.
<code>[options]</code>	Linker options that control how the linker works. Linker options must be preceded by either a slash (/) or a hyphen (-).
<code>startup</code>	A startup code module that arranges the order of the various segments of the program. Failure to link in the correct initialization module usually results in a long list of error messages telling you

	that certain identifiers are unresolved, or that no stack has been created.
myobjs	The .OBJ files you want linked. Specify the path if the files aren't in the current directory. (The linker appends an .OBJ extensions if no extension is present.)
[exe]	The name you want given to the executable file output file. If you don't specify an executable file name, PLINK derives the name of the executable by appending .EXE to the first object file name listed. (The linker assumes or appends .EXE extensions for executable files if no extension is present.)
[mapfile]	Is the name you want given to the map file. If you don't specify a name, the map file name is given the same as <i>exe</i> file (but with the .MAP extension). (The linker appends a .MAP extension if no extension is present.)
[libraries]	The library files you want included at link time. Do not use commas to separate the libraries listed. If a file is not in the current directory or the search path (see the /L option) then you must include the path in the link statement. (The linker appends a .LIB extension if no extension is present.)

The order in which you list the libraries is very important; be sure to use the order defined in this list:

- List any of your own user libraries, noting that if a function is defined more than once, the linker uses the first definition encountered
- Math libraries (if needed)
- Run-time libraries associated with your memory model and platform

PLINK.CFG file

PLINK uses a configuration file called PLINK.CFG for options that you would normally type at the command line (note that configuration files can contain only options, not file names). Configuration files let you save options you use frequently, so you do not have to continually retype them.

PLINK looks for PLINK.CFG in the current directory, then in the directory from which it was loaded.

The following PLINK.CFG file tells PLINK to:

- Look for libraries first in the directory C:\PARADIGM\LIB
- Include debug information in the executables it creates
- Create a detailed segment map.

PLINK.CFG

```
/Lc:\PARADIGM\LIB
/v /s
```



If you specify command-line options in addition to those recorded in a configuration file, the command-line options override any conflicting configuration options.

Linker response files

You can use *response files* with the command-line linkers to specify linker options.

Response files are ASCII files that list linker options and file names that you would normally type at the command line. Response files allow you longer command lines than most operating systems support, plus you don't have to continually type the same information. Response files can include the same information as configuration files, but they also support the inclusion of file names.

Unlike the command line, a response file can be several lines long. To specify an added line, end a line with a plus character (+) and continue the command on the next line. Note that if a line ends with an option that uses the plus to turn it on (such as /v+), the + is not treated as a line continuation character (to continue the line, use /v+ +).

If you separate command-line components (such as .OBJ files from .LIB files) by lines in a response file, you must leave out the comma used to separate them on the command line. For example,

```
/c c0s+
myprog,myexe +
mymap +
mylib cs
```

leaves out the commas you'd have to type if you put the information on the command line:

```
PLINK /c c0s myprog,myexe,mymap,mylib cs
```

To use response files,

1. Type the command-line options and file names into an ASCII text file and save the file. Response files shipped with Paradigm C++ have the .RSP extension.
2. Type

```
PLINK @[path]RESFILE.RSP
```

- where RESFILE.RSP is the name of your response file.

You can specify more than one response file as follows:

```
plink /c @listobjs.rsp,myexe,mymap,@listlibs.rsp
```



You can add comments to response files using semicolons; the linker ignores any text on a line that follows a semicolon.

Using PLINK with PCC.EXE

You can pass options and files to PLINK through the command-line compiler (PCC.EXE) by typing file names on the command line with explicit .OBJ and .LIB extensions. For example,

```
PCC mainfile.obj sub1.obj mylib.lib
```

links MAINFILE.OBJ, SUB1.OBJ, and MYLIB.LIB to produce the executable MAINFILE.EXE.



By default, PCC starts PLINK with the files C0S.OBJ and CS.LIB (initialization module, and run-time library). In addition, the compiler always passes the linker the /c (case-sensitive link) option.

Paradigm C++ tools overview

Paradigm C++ includes many tools to help you create C++ programs. While you can access many of these tools through the Paradigm C++ IDE, you can also run the tools from the command line.

The following table lists the Paradigm tools that come with your Paradigm C++ package:

Table 6-1
Paradigm C++
tools

File	Description
32RTM.EXE	32-bit runtime manager
CAPDOS32.EXE	Utility used by the Paradigm C++ IDE to interface with transfer macros
CPP.EXE	C preprocessor (16-bit)
GREP.COM	File search utility
MAKE.EXE	Make utility
MAKER.EXE	Real-mode MAKE utility
MAKESWAP.EXE	Creates swap file to use with 32-bit command-line tools
OBJXREF.EXE	Utility to examine contents of .OBJ and .LIB files
PASM.EXE	Paradigm assembler
PCC.EXE	Paradigm C++ 16-bit command-line compiler
PCW.EXE	The Paradigm C++ IDE
PDADDREG.EXE	Enables, disables, installs and deletes PCW add-on .DLLs
PLIB.EXE	Utility for maintaining static-link libraries
PLINK.EXE	Paradigm C++ 16-bit linker
RTM.EXE	16-bit runtime manager
TOUCH.COM	Change files stamps to current date/time

Running the command-line tools

Many Paradigm command-line tools (such as the command-line compiler) use DPMI (DOS Protected Mode Interface) to run in protected mode. Protected mode tools run on 80386 and greater machines with at least 640K conventional RAM and at least 4MB extended memory.

Although the compilers run in protected mode, they generate applications that run in real mode. Protected-mode tools have the advantage that they can access more memory than real-mode tools. This helps to compile large projects at faster speeds, without the cost of extensive disk-swapping.

Memory and MAKESWAP.EXE

MAKESWAP applies to DOS only, not to DOS boxes opened under Windows.

If you get “Out of Memory” errors from DOS when running Paradigm command-line tools (or if you have 8MB of RAM and are running the 32-bit command-line tools), create a swap file with the MAKESWAP utility. Describe the size of the swap file in kilobytes. For example, the following command creates a 12MB swap file:

```
MAKESWAP 12000
```

In addition, MAKESWAP supports the following syntax:

MAKESWAP 12M

Both commands create a 12MB swap file in the current directory (named EDPMI.SWP) which the Paradigm command-line tools use when they need additional memory. To enable the swap file, use the DPMI32 environment variable at the DOS prompt, or add this line to your AUTOEXEC.BAT file:

```
set DPMI32=SWAPFILE <SwapFilePath>EDPMI.SWP
```



You must clear the DPMI32 environment variable before you use any 16-bit DPMI-hosted tools with the following command:

```
set DPMI32=
```

The run-time manager and tools

The Paradigm C++ protected-mode tools (such as PCC) use the run-time managers RTM.EXE and 32RTM.EXE. The tools that use run-time managers first load the run-time manager, then do their work, and then unload the run-time manager. If you're accessing 32-bit command-line tools that use the run-time manager many times over a short period (such as from a makefile), you could speed up the process by loading the run-time manager once, calling the tools, then unloading the run-time manager. To load the run-time manager, type 32RTM at the command line. To unload 32RTM, type 32RTM -u.

By default, the run-time manager consumes all available memory when it loads. It then allocates memory to its clients when they request it through the memory manager API routines.

When running in a DOS box under Windows, the amount of memory that RTM reserves is limited to the XMS Memory KB Limit setting for the DOS box. The Property setting for your DOS box should set XMS Memory KB Limit to at least 1024. This value sets the limit on the amount of memory that RTM takes for the 16-bit DOS extended memory application.

Using MAKE

MAKE.EXE is a command-line utility that helps you manage project compilation and link cycles. MAKE helps you quickly build projects by compiling only the files you have modified since the last compilation. In addition, you can set up rules that specify how MAKE should deal with the special circumstances in your builds.

This chapter covers the following topics:

- MAKE basics
- Makefile contents
- Using explicit and implicit rules
- Using MAKE macros
- Using MAKE directives

MAKE basics

MAKE uses rules you write along with its default settings to determine how it should compile the files in your project. For example, you can specify when to build your projects with debug information and to compile your .OBJ files only if the date/time stamps of a source file is more recent than the .OBJ itself. If you need to force the compilation of a module, use TOUCH.EXE to modify the time stamp of one of the module's dependents.

In an ASCII *makefile*, you write explicit and implicit rules to tell MAKE how to treat the files in your project; MAKE determines if it should execute a command on a file or set of files using the rules you set up. Although your commands usually tell MAKE to compile or link a set of files, you can specify nearly any operating system command with MAKE.

The general syntax for MAKE is

```
MAKE [options...] [target[targets]]
```

where *options* are MAKE options that control how MAKE works and *targets* are the names of the files in the makefile that you want to build.

If you need to compile in real mode, use the program MAKER.EXE.

You must separate the MAKE command and the *options* and *target* arguments with spaces. When specifying *targets*, you can use wildcard characters (such as * and ?) to indicate multiple files. To get command-line help for MAKE, type MAKE -?.

Default MAKE actions

When you issue a MAKE command, MAKE looks in the current directory for the file BUILTINS.MAK, which contains the default rules for MAKE (use the -r option to ignore this set of default rules). After loading BUILTINS.MAK, MAKE looks for a file called MAKEFILE or MAKEFILE.MAK (use the -f option to specify a file other than MAKEFILE). MAKE looks for these files first in the current directory, then in the directory where MAKE.EXE is stored. If MAKE can't find either of these files, it generates an error message.

1. After loading the makefile, MAKE tries to build only the first target listed in the makefile by checking the time and date of the dependent files of the first target. If the dependent files are more recent than the target file, MAKE executes the commands to update the target.
2. If one of the first target's dependent files is a target elsewhere in the makefile, MAKE checks that target's dependencies and builds it before building the first target. This chain reaction is called a *linked dependency*.
3. If something during the build process fails, MAKE deletes the target file it was building. Use the **.precious** directive if you want MAKE to keep a target after a build fails.

You can stop MAKE after issuing the make command by pressing *Ctrl+Break* or *Ctrl+C*.

To place MAKE instructions in a file other than MAKEFILE, see the section titled "MAKE options."

BUILTINS.MAK

The file BUILTINS.MAK contains standard rules and macros that MAKE uses when it builds the targets in a makefile. To ignore this file, use the **-r** MAKE option.

Here is the default text of BUILTINS.MAK:

```
#
# <Default ¶ Font>Paradigm C++ - © Copyright 1997 by Paradigm Systems
#

# default is to target 16BIT

CC          = pcc
AS          = psm
.asm.obj:
$(AS) $(AFLAGS) $&.asm
.c.exe:
$(CC) $(CFLAGS) $&.c
.c.obj:
$(CC) $(CFLAGS) /c $&.c
.cpp.exe:
$(CC) $(CFLAGS) $&.cpp
.cpp.obj:
$(CC) $(CPPFLAGS) /c $&.cpp

.SUFFIXES: .exe .obj .asm .c

!if !$d(PARADIGMEXAMPLEDIR)
PARADIGMEXAMPLEDIR = $(MAKEDIR)\..\EXAMPLES
!endif
```

Using TOUCH

TOUCH.EXE updates a file's date stamp so that it reflects your system's current time and date.

Sometimes you might need to force a target to be recompiled or rebuilt even though you haven't changed its source files. One way to do this is to use the TOUCH utility to update

the time stamp of one or more of the target's dependency files. To touch a file (or files), type the following at the command prompt:

```
touch [options] filename [filename...]
```

Because TOUCH is a 32-bit executable, it accepts long file names. In addition, you can use file names that contain the wildcard characters * and ? to "touch" more than a single file at a time.



Before you use TOUCH, make sure your system's internal clock is set correctly.

TOUCH.EXE supports several command-line options:

Table 7-1
TOUCH options

Option	Description
dmm-dd-yy	Sets the date of the file to the specified date
ffilename	Sets the time and date of files to match those of <i>filename</i>
h	Displays help information (same as typing TOUCH without options or file names)
thh:mm:ss	Sets the time of the file to the specified time
v	Verbose mode, shows each file TOUCHed

MAKE options

Use the **-W** option to set default MAKE options.

You can use command-line options to control the behavior of MAKE. MAKE options are case-sensitive and must be preceded with either a hyphen (-) or slash (/). For example, to use a file called PROJECTA.MAK as the makefile, type `MAKE -fPROJECTA.MAK`. Many of the command-line options have equivalent directives that you can use within the makefile.

Table 7-2
MAKE options

Option	Description
-a	Checks dependencies of include files and nested include files associated with .OBJ files and updates the .OBJ if the .h file changed. See also -c .
-B	Builds all targets regardless of file dates.
-c	Caches autodependency information, which can improve MAKE's speed. Use with -a . Do not use this option if MAKE modifies include files (which can happen if you use TOUCH in the makefile or if you create header or include files during the MAKE process).
-Dmacro	Defines macro as a single character, causing an expression !ifdef macro written in the makefile to return true.
[-D]macro=[string]	Defines macro as string. If string contains any spaces or tabs, enclose string in quotation marks. The -D is optional.
-ddirectory	Use this option with -S to specify the drive and directory that MAKER (the real mode version of MAKE) uses when it swaps out of memory. MAKE ignores this option.
-e	Ignores a macro if its name is the same as an environment variable (MAKE uses the environment variable instead of the macro).
-ffilename	Uses filename or filename.MAK instead of MAKEFILE (a space after -f is optional).
-h or -?	Displays MAKE options. Default settings are shown with a trailing plus sign.
-Idirectory	MAKE searches for include files in the current directory first, then in directory you specify with this option.
-i	MAKE ignores the exit status of all programs run from the makefile and continues the build process.

-K	Keeps temporary files that MAKE creates (MAKE usually deletes them). See also “KEEP,” page 7-177. This may be helpful during debugging of your makefiles.
-m	Displays the date and time stamp of each file as MAKE processes it.
-N	Causes MAKE to mimic Microsoft’s NMAKE.
-n	Prints the MAKE commands but does not perform them, this is helpful for debugging makefiles.
-p	Displays all macro definitions and implicit rules before executing the makefile.
-q	Returns 0 if the target is up-to-date and nonzero if it is not (for use with batch files).
-r	Ignores any rules defined in BUILTINS.MAK.
-S	Swaps MAKER out of memory while commands are executed, reducing memory overhead and allowing compilation of large modules. MAKE ignores this option.
-s	Suppresses onscreen command display.
-Umacro	Undefines the previous macro definition of macro.
-W	Writes the specified non-string options to MAKE.EXE, making them defaults.

Setting default MAKE options

The **-W** option lets you set the default options for MAKE. Use the following syntax to set the default options:

```
make [-option[-] ...] -W
```

For example, you could type `MAKE -m -W` to turn the **-m** option on by default (which causes MAKE to always display file dates and times). When you use the **-W** option, MAKE asks you to write changes to MAKE.EXE. Type `Y` to accept the new defaults. To turn off an option that’s on by default, follow the option with a hyphen. For example, to undo the **-m** option change, type

```
MAKE -m- -W
```

Warning! The **-W** option doesn’t work with the following MAKE options:

-Dmacro	-Dmacro=string
-ddirectory	-Usymbol
-ffilename	-? or -h
-Idirectory	



If you attempt to use the **-W** option when the DOS SHARE program is loaded, MAKE displays the message `Fatal: unable to open file MAKE.EXE`.

Compatibility with Microsoft’s NMAKE

Use the **-N** option if you want to use a makefile that was originally created for Microsoft’s NMAKE. The following changes occur when you use **-N**:

- The **\$d** macro is treated differently-use **!ifdef** or **!ifndef** instead.
- Macros that return paths won’t return the last `\`. For example, if `$(<D)` normally returns `C:\CPP\`, the **-N** option causes MAKE to return `C:\CPP`.
- Unless there is a matching **.suffixes** directive, MAKE begins searching for rules from the bottom of the makefile and works its way to the top.
- In implicit rules, MAKE expands **\$*** macros to the target name instead of to the dependent name.

- MAKE interprets the << operator as if it were the && operator; MAKE uses temporary files as response files. These files are then deleted. To keep a file, either use the **-K** MAKE command-line option or use **KEEP** in the makefile.

MAKE usually deletes temporary files it creates.

```
<<FileName.Ext
text
...
<<KEEP
```

If you don't want to keep a temporary file, type **NOKEEP** or type only the temporary (optional) file name. If you don't type a file name, MAKE creates a name for you. If you use **NOKEEP**, it will override the **-K** command-line option.

Using makefiles

A *makefile* is an ASCII file that contains the set of instructions that MAKE uses to build a certain project. Although MAKE assumes your makefile is called MAKEFILE or MAKEFILE.MAK, you can specify a different makefile name with the **-f** option (see page 7-175).

MAKE either builds the target(s) you specify at the MAKE command or it builds *only* the first target it finds in the makefile (to build more than one target, see the section "Symbolic targets"). Makefiles can contain:

- Comments
- Explicit rules
- Implicit rules
- Macros
- Directives

Symbolic targets

A *symbolic target* forces MAKE to build multiple targets in a makefile. When you specify a symbolic target, the dependency line lists all the targets you want to build (a symbolic target basically uses linked dependencies to build more than one target).

For example, the following makefile uses the symbolic target *allFiles* to build both FILE1.EXE and FILE2.EXE:

```
The AllFiles      AllFiles: file1.exe file2.exe
target has no     file1.exe: file1.obj
commands.         pcc file1.obj
                  file2.exe: file2.obj
                  pcc file2.obj
```

Rules for symbolic targets

Observe the following rules when you use symbolic targets:

- Do not type a line of commands following the symbolic target line.
- A symbolic target must have a unique name; it cannot be the name of a file in your current directory.
- Symbolic target names must follow the operating system rules for naming files.

Explicit and implicit rules

You write explicit and implicit rules to instruct MAKE how build the targets in your makefile. In general, these rules are defined as follows:

- Explicit rules are instructions for specific files.
- Implicit rules are general instructions for files that don't have explicit rules.

All the rules you write follow this general format:

```
Dependency line
  Commands
  ?
```

While the dependency line uses a different syntax for explicit and implicit rules, the command line syntax stays the same for both rule types. For more information on linked dependencies see page 7-174.

MAKE supports multiple dependency lines for a single target, and a single target can have multiple command lines. However, only one dependency line should contain a related command line. For example:

```
Target1: dependent1 dep2 dep3 dep4 dep5
Target1: dep6 dep7 dep8
        pcc -c $**
```

Explicit rule syntax

Explicit rules specify the instructions that MAKE must follow when it builds specific targets. Explicit rules name one or more targets followed by one or two colons. One colon means one rule is written for the target(s); two colons mean that two or more rules are written for the target(s).

Explicit rules follow this syntax:

```
target [target...]:[:][path] [dependent[s]...]
  [commands]
  ?
```

<i>target</i>	The name and extension of the file to be built (a <i>target</i> must begin a line in the makefile - you cannot precede the target name with spaces or tabs). To specify more than one target, separate the target names with spaces or tabs. Also, you cannot use a target name more than once in the target position of an explicit rule.
<i>path</i>	A list of directories that tells MAKE where to find the dependent files. Separate multiple directories with semicolons and enclosed the entire path specification in braces.
<i>dependent</i>	The file (or files) whose date and time MAKE checks to see if it is newer than <i>target</i> . Each dependent file must be preceded by a space. If a dependent appears elsewhere in the makefile as a target, MAKE updates or creates that target before using the dependent in the original target (this is known as a <i>linked dependency</i>).
<i>commands</i>	Any operating system commands. You must indent the command line by at least one space or tab, otherwise they are interpreted as a target. Separate multiple commands with spaces (see the section on commands, page 7-180)

If a dependency or command line continues on the following line, use a backslash (\) at the end of the first line to indicate that the line continues. For example,

```
MYSOURCE.EXE: FILE1.OBJ\      #Dependency line
                FILE3.OBJ      #Dependency line continued
                pcc file1.obj file3.obj #Command line
```

Single targets with multiple rules

A single target can have more than one explicit rule. To specify more than a single explicit rule, use a double colon (::) after the target name. The following example shows targets with multiple rules and commands.

```
cpp.obj:
    pcc -c -ncobj $<

.asm.obj:
    tasm /mx $<, asmobj\

mylib.lib :: f1.obj f2.
    echo Adding C files
    plib mylib -+cobjf1 -+cobjf2

mylib.lib :: f3.obj f4.obj
    echo Adding ASM files
    plib mylib -+asmobjf3 -+asmobjf4
```

Implicit rule syntax

An implicit rule specifies a general rule for how MAKE should build files that end with specific file extensions. Implicit rules start with either a path or a period. Their main components are file extensions separated by periods. The first extension belongs to the dependent, the second to the target.

If implicit dependents are out-of-date with respect to the target, or if the dependents don't exist, MAKE executes the commands associated with the rule. MAKE updates explicit dependents before it updates implicit dependents.

Implicit rules follow this basic syntax:

```
[source_dir].source_ext[target_dir].target_ext:
    [commands]
```

- | | |
|---------------------|---|
| <i>{source_dir}</i> | The directory (or directories) where MAKE can find the dependent files. Separate multiple directories with a semicolon. |
| <i>.source_ext</i> | The dependent filename extension. |
| <i>{target_dir}</i> | The directory where MAKE places the target files. Separate multiple directories with a semicolon. |
| <i>.target_ext</i> | The target filename extension. Macros are allowed here. |
| <i>:</i> | Marks the end of the dependency line. |
| <i>commands</i> | Any operating system command or commands. You must indent the command line by at least one space or tab, otherwise they are interpreted as a target. Separate multiple commands with spaces (see the section on commands, page 7-180) |

If two implicit rules match a target extension but no dependent exists, MAKE uses the implicit rule whose dependent's extension appears first in the **.SUFFIXES** list. See “suffixes,” page 7-188.

Explicit rules with implicit commands

See page 7-183 for information on default macros.

A target in an explicit rule can get its command line from an implicit rule. The following example shows an implicit rule followed by an explicit rule without a command line.

```
.c.obj:
    pcc -c $< #This command uses a macro $< described
                later

myprog.obj: #This explicit rule uses the command: pcc
            -c myprog.c
```

The implicit rule command tells MAKE to compile MYPROG.C (the macro \$< replaces the name myprog.obj with myprog.c).

Command syntax

Commands immediately follow an explicit or implicit rule and must begin on a new line with a space or tab.

Commands can be any operating system command, but they can also include MAKE macros, directives, and special operators that your operating system won't recognize (however, note that | can't be used in commands). Here are some sample commands:

```
cd..

pcc -c mysource.c

COPY *.OBJ C:PROJECTA

pcc -c $(SOURCE) #Macros are explained later in the
                  chapter.
```

Commands follow this general syntax:

```
[prefix...] commands
```

Command prefixes

Commands in both implicit and explicit rules can have prefixes that modify how MAKE treats the commands. Table 7.3 lists the prefixes you can use in makefiles:

Table 7-3
Command
prefixes

Prefix	Description
@	Don't display the command while it's being executed.
-num	Stop processing commands in the makefile when the exit code returned from command exceeds the integer num. Normally, MAKE aborts if the exit code is nonzero. No white space is allowed between - and num.
-	Continue processing commands in the makefile, regardless of the exit codes they return.
&	Expand either the macro \$**, which represents all dependent files, or the macro \$?, which represents all dependent files stamped later than the target. Execute the command once for each dependent file in the expanded macro.

Using @

The following command uses the @ prefix, which prevents MAKE from displaying the command onscreen.

```
diff.exe : diff.obj
          @pcc diff.obj
```

Using -num and -

The **-num** and **-** prefixes control the makefile processing when errors occur. You can choose to continue with the MAKE process if an error occurs or you can specify a number of errors to tolerate.

In the following example, MAKE continues processing if PCC returns errors:

```
target.exe : target.obj
target.obj : target.cpp
            pcc -c target.cpp
```

Using &

The **&** prefix issues a command once for each dependent file. It is especially useful for commands that don't take a list of files as parameters. For example,

```
copyall : file1.cpp file2.cpp
         &copy $** c:\temp
```

results in COPY being invoked twice as follows:

```
copy file1.cpp c:\temp
copy file2.cpp c:\temp
```

Without the **&** modifier, MAKE would call COPY only once.

Command operators

While you can use any operating system command in a MAKE command section, you can also use special operators. MAKE supports the normal operators (such as +, -, and so on) as well as the following special operators:

Table 7-4
Command
operators

Operator	Description
<	Use input from a specified file rather than from standard input
>	Send the output from command to file
>>	Append the output from command to file
<<	Create a temporary inline file and use its contents as standard input to command
&&	Create a temporary response file and insert its name in the makefile
delimiter	Use delimiters with temporary response files. You can use any character other than # as a delimiter. Use << and && as a starting and ending delimiter for a temporary file. Any characters on the same line and immediately following the starting delimiter are ignored. The closing delimiter must be written on a line by itself.

Debugging with temporary files

MAKE can create temporary response files when your command lines become too long to place on a single line.

To begin writing to a response file, place the MAKE operator **&&** followed by a delimiter of your choice (| makes a good delimiter) in the makefile. To finish writing to the file, repeat your delimiter.

The following example shows `&&|` instructing MAKE to create a file for the input to PLINK.

```
prog.exe: A.obj B.obj
    PLINK /c @&&|      # &&| opens temp file, @ for PLINK
    c0s.obj $**
    prog.exe
    prog.map
    maths.lib cs.lib
    |      # | closes temp file, must be on first column
```

The response file created by `&&|` contains these instructions:

```
c0s.obj a.obj b.obj
prog.exe
prog.map
maths.lib cs.lib
```

MAKE names temporary file starting at MAKE0000.@@@, where the 0000 increments by one with each temporary file you create. MAKE then deletes the temporary file when it terminates.

Using MAKE macros

A macro is a variable that MAKE expands into a string whenever MAKE encounters the macro in a makefile. For example, you can define a macro called LIBNAME that represents the string “mylib.lib.” To do this, type the line `LIBNAME = mylib.lib` at the beginning of your makefile. Then, when MAKE encounters the macro `$(LIBNAME)`, it substitutes the string `mylib.lib`. Macros let you create template makefiles that you can change to suit different projects.

To use a macro in a makefile, type `$(MacroName)` where *MacroName* is a defined macro. You can use braces or parentheses to enclose *MacroName*.

MAKE expands macros at various times depending on where they appear in the makefile:

- Nested macros are expanded when the outer macro is invoked.
- Macros in rules and directives are expanded when MAKE first looks at the makefile.
- Macros in commands are expanded when the command is executed.

If MAKE finds an undefined macro in a makefile, it looks for an operating-system environment variable of that name (usually defined with **SET**) and uses its definition as the expansion text. For example, if you wrote `$(PATH)` in a makefile and never defined *PATH*, MAKE would use the text you defined for *PATH* in your AUTOEXEC.BAT. See your operating system manuals for information on defining environment variables.

Defining MAKE macros

The general syntax for defining a macro in a makefile is:

```
MacroName = expansion_text.
```

- *MacroName* is case-sensitive (MACRO1 is different from Macro1).
- *MacroName* is limited to 512 characters.
- *expansion_text* is limited to 4096 characters. Expansion characters may be alphanumeric, punctuation, or whitespace.

You must define each macro on a separate line in your makefile and each macro definition must start on the first character of the line. For readability, macro definitions are usually put at the top of the makefile. If MAKE finds more than one definition for *macroName*, the new definition overwrites the old one.

You can also define macros using the **-D** command-line option (see page 7-175). No spaces are allowed before or after the equal sign (=), however, you can define more than one macro can by separating the definitions with spaces. The following examples show macros defined at the command line:

```
make -Dsourcedir=c:projecta
make -Dcommand="pcc -c"
make -Dcommand=pcc option=-c
```



Macros defined in makefiles overwrite those defined on the command line. The following differences in syntax exist between macros entered on the command line and macros written in a makefile.

Table 7-5
Command line
vs. makefile
macros

Syntax	Makefile	Command line
Spaces allowed before and after =	Yes	No
Spaces allowed before <i>macroName</i>	No	Yes

String substitutions in MAKE macros

MAKE lets you temporarily substitute characters in a previously defined macro. For example, if you defined the macro

```
SOURCE = f1.cpp f2.cpp f3.cpp
```

you could substitute the characters .obj for the characters .cpp by using the make command `$(SOURCE:.cpp=.obj)`. This substitution does not redefine the macro.

Rules for macro substitution:

- Syntax: `$(MacroName:original_text=new_text)`
- No whitespace before or after the colon
- Characters in *original_text* must exactly match the characters in the macro definition (text is case-sensitive)

MAKE also lets you use macros within substitution macros. For example,

```
MYEXT=.C
SOURCE=f1.cpp f2.cpp f3.cpp
$(SOURCE:.cpp=$(MYEXT))    #Changes f1.cpp to f1.C, etc.
```

Default MAKE macros

MAKE contains several default macros you can use in your makefiles. Table 7.6 lists the macro definition and what it expands to in explicit and implicit rules.

Table 7-6
Default macros

Macro	Expands in implicit	Expands in explicit
\$*	path\dependent file	path\target file
\$<	path\dependent file+ext	path\target file+ext
\$:	path for dependents	path for target
\$.	dependent file+ext	target file + ext

\$&	dependent file	target file
\$@	path\target file+ext	path\target file+ext
**	path\dependent file+ext	all dependents file+ext
\$?	path\dependent file+ext	old dependents

Table 7-7
Other default
macros

Macro	Expands to	Comment
__MSDOS__	1	If running under DOS
__MAKE__	0x0370	MAKE's hex version number
MAKE	make	MAKE's executable file name
MAKEFLAGS	options	The options typed at the command line
MAKEDIR	directory	Directory where MAKE.EXE is located
PCPPROOT		Will be defined to be the Paradigm C++ root directory if this can be determined by MAKE.



If PCPPROOT is defined, you will find the following BIN, INCLUDE, and LIB directories:

```
$(PCPPROOT)\BIN
$(PCPPROOT)\INCLUDE
$(PCPPROOT)\LIB
```

Modifying default MAKE macros

When the default macros listed in Table 7.6, page 7-183 don't give you the exact string you want, macro modifiers let you extract parts of the string to suit your purpose.

To modify a default macro, use this syntax:

```
$(MacroName [modifier])
```

Table 7.8 lists macro modifiers and provides examples of their use.

Table 7-8
Filename macro
modifiers

Modifier	Part of file name expanded	Example	Result
D	Drive and directory	\$(<D)	C:\PROJECTA\
F	Base and extension	\$(<F)	MYSOURCE.C
B	Base only	\$(<B)	MYSOURCE
R	Drive, directory, and base	\$(<R)	C:\PROJA\SOURCE

Using MAKE directives

MAKE directives resemble directives in languages such as C and Pascal. In MAKE, they perform various control functions, such as displaying commands onscreen before executing them. MAKE directives begin either with an exclamation point or a period, and the override any options given on the command line.

Table 7-9, page 7-185 lists the MAKE directives and their corresponding command-line options (directives override command-line options). Each directive is described in more detail following the table.

Table 7-9
MAKE directives

Directive	Option	Description
.autodepend	-a	Turns on autodependency checking
.cacheautodepend	-c	Turns on autodependency caching
!elif		Acts like a C else if
!else		Acts like a C else
!endif		Ends an !if , !ifdef , or !ifndef statement
!error		Stops MAKE and prints an error message
!if		Begins a conditional statement
!ifdef		Acts like a C #ifdef , testing whether a given macro has been defined
!ifndef		Acts like a C #ifndef , testing whether a given macro is undefined
.ignore	-i	MAKE ignores the return value of a command
!include		Acts like a C #include , specifying a file to include in the makefile
!message		Prints a message to stdout while MAKE runs the makefile
.noautodepend	-a-	Turns off autodependency checking
.nocacheautodepend	-c-	Turns off autodependency caching
.noIgnore	-i-	Turns off .Ignore
.nosilent	-s-	Displays commands before MAKE executes them
.noswap	-S-	Tells MAKE not to swap itself out of memory before executing a command
.path.ext		Tells MAKE to search for files with the extension .ext in path directories
.precious		Saves the target or targets even if the build fails
.silent	-s	MAKE executes commands without printing them first
.suffixes		Determines the implicit rule for ambiguous dependencies
.swap	-S	Tells MAKE to swap itself out of memory before executing a command
!undef		Clears the definition of a macro. After this, the macro is undefined

.autodepend

Autodependencies are the files that are automatically included in the targets you build, such as the header files included in your C++ source code. With **.autodepend** on, MAKE compares the dates and times of all the files used to build the .OBJ, including the autodependency files. If the dates or times of the files used to build the .OBJ are newer than the date/time stamp of the .OBJ file, the .OBJ file is recompiled. You can use **.autodepend** (or **-a**) in place of forming linked dependencies (see page 7-174 for information on linked dependencies).

!error

The syntax of the **!error** directive is:

```
!error message
```

MAKE stops processing and prints the following string when it encounters this directive:

```
Fatal makefile exit code: Error directive: message
```

Embed **!error** in conditional statements to abort processing and print an error message, as shown in the following example:

```

!if !$d(MYMACRO)
#if MYMACRO isn't defined
!error MYMACRO isn't defined
!endif

```

If MYMACRO isn't defined, MAKE terminates and prints:

```
Fatal makefile 4: Error directive: MYMACRO isn't defined
```

Error-checking controls

MAKE offers four different controls to control error checking:

- The **.ignore** directive turns off error checking for a selected portion of the makefile.
- The **-i** command-line option turns off error checking for the entire makefile.
- The **-num** prefix, which is entered as part of a rule, turns off error checking for the related command if the exit code exceeds the specified number.
- The **-** prefix turns off error checking for the related command regardless of the exit code.

!if and other conditional directives

The **!if** directive works like C if statements. As shown here, the syntax of **!if** and the other conditional directives resembles compiler conditionals:

```

!if condition  !if condition  !if condition  !ifdef macro
!endif          !else          !elif condition !endif
                !endif          !endif

```

The following expressions are equivalent:

```

!ifdef macro and !if $d(macro)
!ifndef macro and !if !$d(macro)

```

These rules apply to conditional directives:

- One **!else** directive is allowed between **!if**, **!ifdef**, or **!ifndef** and **!endif** directives.
- Multiple **!elif** directives are allowed between **!if**, **!ifdef**, or **!ifndef**, **!else** and **!endif**.
- You can't split rules across conditional directives.
- You can nest conditional directives.
- **!if**, **!ifdef**, and **!ifndef** must have matching **!endif** directives within the same file.

The following information can be included between **!if** and **!endif** directives:

- Macro definition
- **!include** directive
- Explicit rule
- **!error** directive
- Implicit rule
- **!undef** directive

Condition in **if** statements represents a conditional expression consisting of decimal, octal, or hexadecimal constants and the operators shown in Table 7-10, page 7-187.

Table 7-10
Conditional
operators

Operator	Description	Operator	Description
-	Negation	?:	Conditional expression
~	Bit complement	!	Logical NOT
+	Addition	>>	Right shift
-	Subtraction	<<	Left shift
*	Multiplication	&	Bitwise AND
/	Division		Bitwise OR
%	Remainder	^	Bitwise XOR
&&	Logical AND	>=	Greater than or equal*
	Logical OR	<=	Less than or equal*
>	Greater than	==	Equality*
<	Less than	!=	Inequality*

*Operator also works with string expressions.

MAKE evaluates a conditional expression as either a 32-bit signed integer or as a character string.

!include

This directive is like the **#include** preprocessor directive for the C or C++ language—it lets you include the text of another file in the makefile:

```
!include filename
```

You can enclose *filename* in quotation marks (“ ”) or angle brackets (< >) and nest directives to unlimited depth, but writing duplicate **!include** directives in a makefile isn’t permitted—you’ll get the error message cycle in the include file.

Rules, commands, or directives must be complete within a single source file; you can’t start a command in an **!include** file, then finish it in the makefile.

MAKE searches for **!include** files in the current directory unless you’ve specified another directory with the **-I** command-line option.

!message

The **!message** directive lets you send messages to the screen from a makefile. You can use these messages to help debug a makefile that isn’t working properly. For example, if you’re having trouble with a macro definition, you could put this line in your makefile:

```
!message The macro is defined here as: $(MacroName)
```

When MAKE interprets this line, it will print onscreen `The macro is defined here as: .CPP` (assuming the macro expands to `.CPP`). Using a series of **!message** directives, you can debug your makefiles.

.path.ext

The **.path.ext** directive tells MAKE where to look for files with a certain extension. The following example tells MAKE to look for files with the `.c` extension in `C:SOURCE` or `C:CFILES` and to look for files with the `.obj` extension in `C:OBJS`.

```
.path.c = C:SOURCE;C:CFILES
.path.obj = C:OBJS
```

.precious

If a MAKE build fails, MAKE deletes the target file. The **.precious** directive prevents the file deletion, which you might desire for certain kinds of targets. For example, if your build fails to add a module to a library, you might not want the library to be deleted.

The syntax for **.precious** is

```
.precious: target [target ...]
```

.suffixes

The **.suffixes** directive tells MAKE the order (by file extensions) for building implicit rules.

The syntax of **.suffixes** is

```
.suffixes: .ext [.ext ...]
```

where **.ext** represents the dependent file extensions you list in your implicit rules. For example, you could include the line `.suffixes: .asm .c .cpp` to tell MAKE to interpret implicit rules beginning with the ones dependent on .ASM files, then .C files, then .CPP files, regardless of what order they appear in the makefile.

The following **.suffixes** example tells MAKE to look for a source file first with an .ASM extension, next with a .C extension, and finally with a .CPP extension. If MAKE finds MYPROG.ASM, it builds MYPROG.OBJ from the assembler file by calling TASM. MAKE then calls PLINK; otherwise, MAKE searches for MYPROG.C to build the .OBJ file or it searches for MYPROG.CPP.

```
.suffixes: .asm .c .cpp

myprog.exe: myprog.obj
plink myprog.obj

.cpp.obj:
    pcc -P $<
.asm.obj:
    tasm /mx $<
.c.obj:
    pcc -P- $<
```

!undef

!undef (undefine) clears the given macro, causing an **!ifdef** *MacroName* test to fail.

The syntax of the **!undef** directive is

```
!undef MacroName
```

Using macros in directives

You can use the **\$d** macro with the **!if** conditional directive to perform some processing if a specific macro is defined. Follow the **\$d** with macro name enclosed in parentheses or braces, as shown in the following example:

```
!if $d(DEBUG)           #If DEBUG is defined,
pcc -v f1.cpp f2.cpp    #compile with debug information;
!else                   #otherwise (else)
```

```
pcc -v- f1.cpp f2.cpp #don't include debug information.  
!endif
```

Null macros

While an undefined macro causes an **!ifdef** *MacroName* test to return false, *MacroName* defined as null will return true. You define a null macro by following the equal sign in the macro definition with either spaces or a return character. For example, the following line defines a null macro in a makefile:

```
NULLMACRO =
```

Either of the following lines can define a null macro on the MAKE command line:

```
NULLMACRO=" "  
-DNULLMACRO
```


PLIB.EXE

PLIB.EXE is a utility that manages libraries of individual .OBJ (object module) files. A library is a convenient tool for dealing with a collection of object modules as a single unit.

This chapter covers the basics of using the PLIB library utility including:

- PLIB options
- Operation list
- Response files
- PLIB examples

PLIB basics

The libraries included with Paradigm C++ were built with the PLIB.EXE library utility. You can use PLIB to build your own libraries, or to modify the Paradigm C++ libraries, your libraries, libraries furnished by other programmers, or commercial libraries you've purchased.

When PLIB modifies an existing library, it always creates a copy of the original library with a .BAK extension.

You can use PLIB to:

- Create a new library from a group of object modules.
- Add object modules or other libraries to an existing library.
- Remove object modules from an existing library.
- Replace object modules from an existing library.
- Extract object modules from an existing library.
- List the contents of a new or existing library.

PLIB can also create (and include in the library file) an extended dictionary, which can be used to speed up linking.

Although PLIB is not essential for creating executable programs with Paradigm C++, it is a useful programming productivity tool that can be indispensable for large development projects.

PLIB options

The PLIB command line takes the following general form, where items listed in square brackets are optional:

```
plib [@respfile] [option] libname [operations] [, listfile]
```

Table 8-1, page 8-192 lists the command-line options available in PLIB. Each is described in detail following the table.



For an online summary of PLIB command-line options, type PLIB and press *Enter*.

Table 8-1
PLIB options

Option	Description
@respfile	The path and name of the response file you want to include. You can specify more than one response file.
libname	The DOS path name of the library you want to create or manage. Every PLIB command must be given a libname. Wildcards are not allowed. PLIB assumes an extension of .LIB if none is given. Use only the .LIB extension because both PCC and the Paradigm C++ IDE require the .LIB extension in order to recognize library files. Note: If the named library does not exist and there are add operations, PLIB creates the library.
/C	The case-sensitive flag. This option is not normally used.
/E	Creates extended dictionary
/Psize	Sets the library page size to size.
/O	Purges comment records.
operations	The list of operations PLIB performs. Operations can appear in any order. If you only want to examine the contents of the library, don't give any operations.
listfile	The name of the file that lists library contents. The listfile name (if given) must be preceded by a comma. No listing is produced if you don't give a file name. The listing is an alphabetical list of each module. The entry for each module contains an alphabetical list of each public symbol defined in that module. The default extension for the listfile is .LST. You can direct the listing to the screen by using the listfile name CON, or to the printer by using the name PRN.

PLIB /C option

Using case-sensitive symbols in a library

When you add a module to a library, PLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add a module to the library that duplicates a symbol, PLIB displays an error message and doesn't add the module.

Normally, when PLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not treated differently (for example, the symbols lookup and LOOKUP are treated as duplicates). You can use the /C option to add a module to a library that includes symbols differing only in case.

Don't use /C if you plan to use the library with other linkers or let other people use the library.

PLIB normally rejects symbols that differ only in case because some linkers aren't case-sensitive. PLINK has no problem distinguishing uppercase and lowercase symbols. As long as you use your library only with PLINK, you can use the PLIB /C option without any problems.

PLIB /E option

Creating an extended dictionary

To increase the capacity of PLINK for large links, you can use PLIB to create an extended dictionary and append it to the library file. This dictionary contains, in a compact form, information that is not included in the standard library dictionary and that lets PLINK process library files so that those modules not needed in the link are not processed.

To create an extended dictionary for a library that is being modified, use the /E option when you start PLIB to add, remove, or replace modules in the library. To create an extended dictionary for an existing library that you don't want to modify, use the /E option. For example, if you type the following text, PLINK appends an extended dictionary to the specific library:

```
plib /E mylib
```

If you get the message “Table limit exceeded”, use /E to see if it helps. If you use /E to add a library module containing a C++ class with a virtual function, you'll get the error message, Library contains COMDEF records—extended dictionary not created.

PLIB /P option

Setting the page size to create a large library

Every DOS library file contains a dictionary that appears at the end of the .LIB file, following all of the object modules. For each module in the library, the dictionary contains a 16-bit address of that particular module within the .LIB file; this address is given in terms of the library page size (it defaults to 16 bytes).

The library page size determines the maximum combined size of all object modules in the library, which cannot exceed 65,536 pages. The default (and minimum) page size of 16 bytes allows a library of about 1 MB in size. To create a larger library, use the /P option to increase the page size. The page size must be a power of 2, and it cannot be smaller than 16 or larger than 32,768.

All modules in the library must start on a page boundary. For example, in a library with a page size of 32 (the lowest possible page size higher than the default 16), an average of 16 bytes will be lost per object module in padding. If you attempt to create a library that is too large for the given page size, PLIB will issue an error message and suggest that you use /P with the next available higher page size.

Using PLIB response files

When you use a large number of operations, or if you find yourself repeating certain sets of operations over and over, you will probably want to use response files. A response file is an ASCII text file (which can be created with the Paradigm C++ editor) that contains all or part of a PLIB command. Using PLIB response files, you can build PLIB commands larger than would fit on one command line. Response files can

- Contain more than one line of text; use the ampersand character (&) at the end of a line to indicate that another line follows.
- Include a partial list of commands. You can combine options from the command line with options in a response file.
- be used with other response files in a single PLIB command line.

Operation list

The operation list describes what actions you want PLIB to do and consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character action symbol followed by a file or module name. You can put whitespace around either the action symbol or the file or module name, but not in the middle of a two-character action or in a name.

You can put as many operations as you like on the command line, up to DOS's COMMAND.COM-imposed line-length limit of 127 characters. The order of the operations is not important. PLIB always applies the operations in a specific order:

To replace a module, first remove it, then add the replacement module. The following shows the order in which PLIB handles these operations:

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.
4. Wildcards are never allowed in file or module names.



See Table 8-3 for more information on Add, Remove and Extract.

PLIB finds the name of a module by stripping any drive, path, and extension information from the given file name.



PLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you need to supply only the module name, not the path and .OBJ extension.

To create a library, add modules to a library that does not yet exist.

PLIB recognizes three action symbols (*, +, &), which you can use singly or combined in pairs for a total of five distinct operations. Table 8-2 summarizes these three action symbols. The order of the characters is not important for operations that use a pair of characters. The action symbols and what they do are listed here:

Table 8-2
PLIB action symbols

Symbol	Name	Description
-*	Extract &	PLIB copies the named module to the corresponding file name and then removes it from the library.
*-	Remove	Removes named module from library.
-+	Replace	PLIB replaces the named module with the corresponding file.

Table 8-3
PLIB operations

Option	Description
Add	<p>PLIB adds the named file to the library. If the file has no extension, PLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library.</p> <p>If a module being added already exists, PLIB displays a message and does not add the new module.</p>
Remove	<p>PLIB removes the named module from the library. If the module does not exist in the library, PLIB displays a message.</p> <p>A remove operation needs only a module name. PLIB lets you enter a full path name with drive and extension included, but ignores everything except the module name.</p>
Extract	<p>PLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, PLIB displays a message and does not create a file. If the named file already exists, it is overwritten.</p> <p>You can't directly rename modules in a library. To rename a module, extract and remove it, rename the file just created, then add it back into the library.</p>

PLIB examples

These simple examples demonstrate some of the different things you can do with PLIB:

Example 1

To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type:

```
plib mylib +x +y +z.
```

Example 2

To create a library named MYLIB.LIB and get a listing in MYLIB.LST too, type:

```
plib mylib +x +y +z, mylib.lst.
```

Example 3

To get a listing in CS.LST of an existing library CS.LIB, type:

```
plib cs, cs.lst.
```

Example 4

To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type:

```
plib mylib -x +a -z.
```

Example 5

To extract module Y.OBJ from MYLIB.LIB and get a listing in MYLIB.LST, type:

```
plib mylib *y, mylib.lst.
```

Example 6

To create a new library named ALPHA, with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:

1. First create a text file, ALPHA.RSP, with

```
+a.obj +b.obj +c.obj &  
+d.obj +e.obj +f.obj &  
+g.obj
```

2. Then use the PLIB command, which produces a listing file named ALPHA.LST:

```
lib alpha @alpha.rsp, alpha.lst
```


Exception handling

This chapter describes the Paradigm C++ error-handling mechanisms generally referred to as *exception handling*. The Paradigm C++ implementation of C++ exception handling is consistent with the proposed ANSI specification. The exception-handling mechanisms that are available in C programs are referred to as *structured exceptions*. Paradigm C++ provides full compiling, linking, and debugging support for C programs with structured exceptions. See the section “C-based structured exception,” page 9-204, and “,” page 9-203 for a discussion of compiler options for programming with exceptions.

C++ exception handling

The C++ language defines a standard for exception handling. The standard insures that the power of object-oriented design is supported throughout your program.

In accordance with the ANSI/ISO C++ working paper specification, Paradigm C++ supports the termination exception-handling model. When an abnormal situation arises at run-time, the program could terminate. However, throwing an exception allows you to gather information at the throw point that could be useful in diagnosing the causes that led to failure. You can also specify in the exception handler the actions to be taken before the program terminates. Only synchronous exceptions are handled, meaning that the cause of failure is generated from within the program. An event such as *Ctrl-C* (which is generated from outside the program) is not considered to be an exception.

C++ exceptions can be handled only in a **try/catch** construct.

Syntax:

```

try-block:
    try compound-statement handler-list

handler-list:
    handler handler-list opt

handler:
    catch (exception-declaration) compound-statement

exception-declaration:
    type-specifier-list declarator
    type-specifier-list abstract-declarator
    type-specifier-list
    ...

throw-expression:
    throw assignment-expression opt
  
```



The **catch** and **throw** keywords are not allowed in a C program.



The *try-block* is a statement that specifies the flow of control as the program executes. The try-block is designated by the **try** keyword. After the keyword, braces surround a program block that can generate exceptions. The language structure specifies that any

exceptions that occur should be raised within the *try-block*. See "statements" in the online Help index for more information.

The handler is a block of code designed to handle an exception. The C++ language requires that at least one handler be available immediately after the *try-block*. There should be a handler for each exception that the program can generate.

When the program encounters an abnormal situation for which it is not designed, you can transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception.

The exception-handling mechanism requires the use of three keywords: **try**, **catch**, and **throw**. The *try-block* specified by **try** must be followed immediately by the *handler* specified by **catch**. If an exception is thrown in the *try-block*, program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. Failure to do so could result in abnormal termination of the program.

Exception declarations

Although C++ allows an exception to be of any type, it is useful to make exception classes. The exception object is treated exactly the way any object would be treated. An exception carries information from the point where the exception is thrown to the point where the exception is caught. This is information that the program user will want to know when the program encounters some anomaly at run-time.



Predefined exceptions, specified by the C++ language, are documented in the online Help Book Shelf index under "Run-time support", "operator new" or "xalloc". To get to the Book Shelf index, choose Help|Keyboard and click the Book Shelf menu tab. Paradigm C++ provides additional support for exceptions. These extensions are also documented under "classes" in the online Help index.

Throwing an exception

A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the **try** keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

1. The program searches for a matching handler
2. If a handler is found, the stack is unwound to that point
3. Program control is transferred to the handler

If no handler is found, the program will call the *terminate* function. If no exceptions are thrown, the program executes in the normal fashion.

A *throw expression* is also referred to as a throw-point. You can specify whether an exception can be thrown by using one of the following syntax specifications:

```

1. throw throw_expression ;
2. throw ;
3. void my_func1( ) throw (A, B)
   {
       // Body of function.
   }
4. void my_func2 ( ) throw ( )
   {
       // Body of this function.
   }

```

The first case specifies that *throw_expression* is to be passed to a handler.

The second case specifies that the exception currently being handler is to be thrown again. An exception must currently exist. Otherwise, *terminate* is called.



The third case specifies a list of exceptions that *my_func1* can throw. No other exceptions should propagate out of *my_func1*. If an exception other than *A* or *B* is generated within *my_func1*, it is considered to be an unexpected exception and program control will be transferred to the *unexpected* function. By default, the *unexpected* function ends with a call to *abort* but it can throw an exception. For more information, see "unexpected" in the online Help Book Shelf index. The Book Shelf index is accessed by choosing Help|Keyboard and clicking on the Book Shelf menu tab.

The final case specifies that *my_func2* should throw no exceptions. If some other function (for example, **operator new**) in the body of *my_func2* throws an exception, such an exception should be caught and handled within the body of *my_func2*. Otherwise, such an exception is a violation of *my_func2* exception specification. The *unexpected* function is then called.

When an exception occurs, the throw expression initializes a temporary object of the type *T* (to match the type of argument *arg*) used in *throw(T arg)*. Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

Handling an exception

The exception handler is indicated by the **catch** keyword. The handler must be placed immediately after the try-block. The keyword **catch** can also occur immediately after another **catch**. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list. The possible conversions are listed after the try-block syntaxes.

The following syntaxes, following the try-block, are valid:

```

Try {
    // Include any code that might throw an exception
}
1. catch (T X)
   {
       // Take some actions
   }
2. catch ( ... )
   {
       // Take some actions
   }

```

The first statement is specifically defined to handle an object of type *T*. If the argument is *T*, *T&*, *const T*, or *const T&*, the handler will accept an object of type *X* if any of the following are true:

- *T* and *X* are of the same type
- *T* is an accessible base class for *X* in the throw expression
- *T* is a pointer type and *X* is a pointer type that can be converted to *T* by a standard pointer conversion at the throw point

The statement **catch** (...) will handle any exception, regardless of type. This statement, if used, must be the last handler for its try-block.

Every exception thrown by the program must be caught and processed by the exception handler. If the program fails to provide an exception handler for a thrown exception, the program will call *terminate*.

Exception handlers are evaluated in the order that they are encountered. An exception is caught when its type matches the type in the **catch** statement. Once a type match is made, program control is transferred to the handler. The stack will have been unwound upon entering the handler. The handler specifies what actions should be taken to deal with the program anomaly.

A **goto** statement can be used to transfer program control out of a handler but such a statement can never be used to enter a handler or try-block.

After the handler has executed, the program can continue at the point after the last handler for the current try-block. No other handlers are evaluated for the current exception.

Exception specifications

The C++ language makes it possible for you to specify any exceptions that a function can throw. This *exception specification* can be used as a suffix to the function declaration. The syntax for exception specification is as follows:

```
exception-specification:
    throw (type-id-listopt)
type-id-list:
    type-id
    type-id-list, type-id
```

The function suffix is not considered to be part of the function's type. Consequently, a pointer to a function is not affected by the function's exception specification. Such a pointer checks only the function's return and argument types. Therefore, the following is legal:

```
void f2(void) throw() ;           // Should not throw exceptions
void f3(void) throw (BETA) ;     // Should only throw BETA objects
void (* fptr)() ;                // Pointer to a function returning void
fptr = f2 ;
fptr = f3 ;
```

Extreme care should be taken when overriding virtual functions. Again, because the exception specification is not considered part of the function type, it is possible to violate the program design. In the following example, the derived class BETA::vfunc is defined so that it throws an exception – a departure from the original function declaration.

```
Class ALPHA {
public:
    virtual void vfunc(void) throw ( ) { }; // Exception specification
};
class BETA : public ALPHA {
```



```

    struct BETA_ERR { };
    void vfunc(void) throw( BETA_ERR ) { }; // Exception specification
is
                                           // changed
};

```

The following are examples of functions with exception specifications.

```

void f1(); // The function can throw any exception
void f2(); throw (); // Should not throw any exceptions
void f3(); throw ( A, B* ); // Can throw exceptions publicly derived
// from A, or a pointer to publicly
derived B

```

The definition and all declarations of such a function must have an exception specification containing the same set of type-id's. If a function throws an exception not listed in its specification, the program will call unexpected. This is a run-time issue – it will not be flagged at compile time. Therefore, care must be taken to handle any exceptions that can be thrown by elements called within a function.

Example 2

```

// HOW TO MAKE EXCEPTION-SPECIFICATIONS AND HANDLE ALL EXCEPTIONS
#include <iostream.h>

// EXCEPTION DECLARATIONS
class Alpha {
    // Include something that shows why you chose to throw this
    exception.
};
Alpha alpha_inst;

class Beta {
    // Include something that shows why you chose to throw this
    exception.
};
Beta beta_inst;

// THROW ONLY Alpha OR Beta TYPE OBJECTS
void f3(char c) throw (Alpha, Beta) {
    cout << "f3() was called" << endl;
    if (c == 'a').
        throw( alpha_inst );
    if (c == 'b')
        throw( beta_inst );
    else ; // DO NOTHING WITH OTHER CHARACTERS
}

// SHOULD NOT THROW EXCEPTIONS
void f2 (char ch) throw( ) {
    try { // WRAP ALL CODE IN A TRY-BLOCK
        cout << "f2( ) was called" << endl;
        f3(ch);
    }
    // HERE ARE HANDLERS FOR THE EXCEPTIONS WE KNOW COULD BE THROWN
    catch (Alpha& alpha_inst) { cout << "Caught Alpha exception.";}
    catch (Beta& beta_inst) { cout << "Caught Beta exception.";}

    // IF THE CODE IS MODIFIED LATER SO THAT SOME OTHER EXCEPTION IS
    // THROWN, IT IS HANDLED HERE AND WE AVOID VIOLATING THE f2() THROW

```

```

// SPECIFICATION
catch ( ... ) {
    // BUT, WE POST OURSELVES A WARNING MESSAGE.
    cout << "Warning: f2() has elements with exceptions!" << endl;
}

}

int main(void) {
    char trigger;

    try {
        cout << "Input a character:";
        cin >> trigger;
        f2(trigger);
        cout << "\nSuccess.";
        return 0; //WE GET HERE ONLY IF EVERYTHING EXECUTES WELL.
    }
    catch ( ... ) {
        cout << "Need more handlers!";
        return 1;
    }
}

```

Sample output when 'a' is the input

```

Input a character: a
f2() was called
f3() was called
Caught Alpha exception.
Success.

```



If an exception is thrown which is not listed in the exception specification, the *unexpected* function will be called. The following diagrams illustrate the sequence of events that can occur when *unexpected* is called. See "Run-time support" in the online Help Book Shelf index, for a description of the "set_terminate", "set_unexpected", and "unexpected" functions. The Book Shelf index is accessed by choosing Help|Keyboard and clicking on the Book Shelf menu tab.

Program behavior when a function is registered with *set_unexpected()*;

```

unexpected()    // CALLED AUTOMATICALLY

|
|
|    // DEFINE YOUR UNEXPECTED HANDLER
|    unexpected_function my_unexpected( void )
|    {
|        //DEFINE ACTION TO TAKE POSSIBLE MAKE ADJUSTMENTS
|    }
|
|    // REGISTER YOUR HANDLER
|    set_unexpected( my_unexpected );
|
my_unexpected();

```

Program behavior when no function is registered with *set_unexpected()* but there is a function registered with *set_terminate()*:

```

unexpected()    // CALLED AUTOMATICALLY
|
terminate()
|
// DEFINE YOUR TERMINATION SCHEME
terminate_function my_terminate( void )
{
    // TAKE ACTIONS BEFORE TERMINATING
    // SHOULD NOT THROW EXCEPTIONS
    exit(1); // MUST END SOMEHOW.
}

// REGISTER YOUR TERMINATION FUNCTION
set_terminate( my_terminate )

my_terminate()
// PROGRAM ENDS.

```

Constructors and destructors



When an exception is thrown, the copy constructor is called for the thrown value. The copy constructor is used to initialize a temporary object at the throw point. Other copies can be generated by the program. See "copy constructor" in the online Help index for more information.

When program flow is interrupted by an exception, destructors are called for all automatic objects that were constructed since the beginning of the try-block was entered. If the exception was thrown during construction of some object, destructors will be called only for those objects that were fully constructed. For example, if an array of objects was under construction when an exception was thrown, destructors will be called only for the array elements which were already fully constructed.



Destructors are called by default. See "Exception handling/RTTI" ":", for information about exception-handling switches.

When a C++ exception is thrown, the stack is unwound. By default, during stack unwinding, destructors are called by automatic objects. You can use the `-xd` compiler option to switch the default off.

Setting exception handling options

The following command-line options can be used to set exception handling:

Setting	Command-line option
Enable exception handling	-x
Enable destructor cleanup	-xd
Enable throwing exceptions from a DLL	-xds
Enable exception location information	-xp

Unhandled exceptions



If an exception is thrown and no handler is found it, the program will call the *terminate* function. This following diagram illustrates the series of events that can occur when the program encounters an exception for which no handler can be found. See "Run-time support" in the online Help Book Shelf index for a description of the *terminate* function. The Book Shelf index is accessed by choosing Help|Keyboard and clicking on the Book Shelf menu tab.

```

terminate();
|
|
abort();
// PROGRAM ENDS.

```

For portability, you can use the *try* and *except* macros defined in `except.h`.

For try-except exception-handling implementations the syntax is as follows:

_try *compound-statement* (in a C module)
try *compound-statement* (in a C++ module)

__except (*expression*) *compound-statement*

```
try-block:
```

__try *compound-statement*

__finally *compound-statement*

- C structured exceptions can be used in C++ programs.

- The following C exception support functions can be used in a C and C++ programs:

Paradigm C++ does not require that the *UnhandledExceptionFilter* function be used

only in the except filter of `_try/except` or `try/_except` blocks. However, program

behavior is undefined when this function is called outside of the `__try/__except` or `try/__except` block.

Handling C-based exceptions

The full functionality of an `__except` block is allowed in C++. If an exception is generated in a C module, it is possible to provide a handler-block in a separate calling C++ module.

If a handler can be found for the generated structured exception, the following actions can be taken:

- Execute the actions specified by the handler
- Ignore the generated exception and resume program execution
- Continue the search for some other handler (regenerate the exception)

These actions are consistent with the design of structured exceptions. The following example shows how to mix C and C++ exceptions. Note that the C mechanism uses the `try` and `__except` keywords. The C++ mechanism uses the required `try` and `catch` keywords.

```
/* In PROG.C */
void func(void) {
    ...
    /* generate an exception */
    RaiseException( /* specify your arguments */ );
    ...
}

// In CALLER.CPP
// How to test for C++ or C-based exceptions.
#include <excpt.h>
#include <iostream.h>

int main(void) {
    try
    {
        // test for C++ exceptions
        try
        {
            // test for C-based structured exceptions
            func();
        }
        __except( /* filter-expression */ )
        {
            cout << "A structured exception was generated.";
            ...
            /* specify action to take for this structured exception */
            return -1;
        }
        return 0;
    }
    catch ( ... )
    {
        // handler for any C++ exception
        cout << "A C++ exception was thrown.";
        return 1;
    }
}
```

Structured exceptions also allow you to program a termination handler. The termination handler can be used only in a C module and is specified by the `__finally` keyword. The termination handler ensures that the code in the `__finally` block is executed no matter how the flow within the `__try` exits. The `__finally` keyword is not allowed in a C++ program. Consequently, the `__try/ __finally` block is not supported in a C++ program.

Even though the `__try/ __finally` block is not supported in a C++ program, a C-based exception generated by the operating system or the program will still result in proper stack unwinding of objects with destructors. You can use this to emulate a `__finally` block by creating a local object whose destructor does the necessary cleanup. Any module compiled with the `-xd` compiler option (this option is on by default) will have destructors invoked for all objects with **auto** storage. Stack unwinding occurs from the point where the exception is thrown to the point where the exception is caught.



Destructors are called by default. See “Exception handling/RTTI,” page 3-68 for information about exception-handling switches.

Using inline assembly

Inline assembly is assembly-language instructions embedded within your C and C++ code. Inline assembly instructions are compiled and assembled along with your code rather than being assembled in separate assembly modules.

This chapter describes how to use inline assembly with Paradigm C++. The following topics are discussed:

- Inline assembly syntax and usage
 - Using the **asm** keyword to place an assembly instruction within your C/C++ code
 - Using C symbols in your **asm** statement to reference data and functions
 - Using register variables, offsets, and size overrides
 - Using C structure members
 - Using jump instructions and labels
- Using the **-B** compiler option and **#pragma** inline statement to compile inline assembly
- Using the built-in assembler (PASM)



See Paradigm C++ equivalents of command-line options on page 3-112.

Inline assembly syntax and usage

This section describes inline assembly syntax, and how to use inline assembly instructions with C++ structures, pointers, and identifiers.

To place an assembly instruction in your C/C++ code, use the **asm** keyword. The format is

asm *opcode operands* ; or *newline*

where:

- *opcode* is valid 80x86 instruction.
- *operands* contains the operand(s) acceptable to the opcode, and can reference C constants, variables, and labels.
- The end of the **asm** statement is signaled by either ; (semicolon) or by *newline* (a new line).

A new **asm** statement can be placed on the same line, following a semicolon, but no **asm** statement can continue to the next line. To include multiple **asm** statements, surround them with braces. The initial brace must appear on the same line as the **asm** keyword.

Three **asm** statements are shown here; two on one line, and one below them.

```
asm {  
    pop ax; pop ds  
    iret  
}
```

Semicolons are not used to start comments (as they are in PASM). When commenting **asm** statements, use C-style comments, like this:

```
asm mov ax,ds;           /* This comment is OK */
asm {pop ax; pop ds; iret;} /* This comment is also legal */
asm push ds              ;THIS COMMENT IS INVALID!!
```

The assembly-language portion of the statement is copied straight to the output, embedded in the assembly language that Paradigm C++ is generating from your C or C++ instructions. Any C symbols are replaced with appropriate assembly language equivalents.

Each **asm** statement is considered to be a C statement. For example, the following construct is a valid C **if** statement:

```
myfunc( )
{
    int i;
    int x;
    if (i > 0)
        asm mov x,4
    else
        i = 7;
}
```



A semicolon isn't needed after the *move x,4* instruction. **asm** statements are the only statements in C that depend on the occurrence of a new line to indicate that they have ended. Although this isn't in keeping with the rest of the C language, it is the convention adopted by several UNIX-based compilers.

An **asm** statement can be used as an executable statement inside a function, or as an external declaration outside of a function. **asm** statements located inside functions are placed in the code segment, and **asm** statements located outside functions are placed in the data segment.

Inline assembly references to data and functions

You can use any C symbol in your **asm** statements, including automatic (local) variables, register variables, and function parameters. Paradigm C++ automatically converts these symbols to the appropriate assembly-language operands and appends underscores onto identifier names.

In general, you can use a C symbol in any position where an address operand would be legal. Of course, you can use a register variable wherever a register would be a legal operand.

If the assembler encounters an identifier while parsing the operands of an inline-assembly instruction, it searches for the identifier in the C symbol table. The names of the 80x86 registers are excluded from this search. Either uppercase or lowercase forms of the register names can be used.

Inline assembly and register variables

Inline assembly code can freely use SI or DI as scratch registers. If you use SI or DI in inline assembly code, the compiler won't use these registers for register variables.

In 16-bit code BX is available for use as a scratch register. In 32-bit code, the corresponding EBX is not available for use as a scratch register.

When you use PCC32 or PCC32A to compile a C or C++ source file, including files with inline assembly, the compiler preserves the EBX register. However, when you compile an assembly .ASM source file, you are responsible for preserving the EBX register. This is true whether you compile the .ASM source file with a 32-bit compiler or use PASM32.

Inline assembly, offsets, and size overrides

When programming, you don't need to be concerned with the exact offsets of local variables: using the variable name will include the correct offsets.

It might be necessary, however, to include appropriate WORD PTR, BYTE PTR, or other size overrides on assembly instruction. A DWORD PTR override is needed on LES or indirect far call instructions.

Using C structure members

You can reference structure members in an inline-assembly statement in the usual way (that is, with *variable.member*). When you do this, you are working with variables, and you can store or retrieve values in these structure members. However, you can also directly reference the member name (without the variable name) as a form of numeric constant. In this situation, the constant equals the offset (in bytes) from the start of the structure containing that member. Consider the following program fragment:

```
struct myStruct {
    int a_a;
    int a_b;
    int a_c;
} myA ;

myfunc ( )
{
    . . .
    asm {mov ax, WORD PTR myA.a_b
        mov bx, WORD PTR myA.a_c
        }
    . . .
}
```

This fragment declares a structure type named *myStruct* with three members: *a_a*, *a_b*, and *a_c*. It also declares a variable *myA* of type *myStruct*. The first inline-assembly statement moves the value contained in *myA.a_b* into the register AX. The second moves the value at the address *[di] + offset(a_c)* into the register BX (it takes the address stored in DI and adds to it the offset of *a_c* from the start of *myStruct*). In this sequence, these assembler statements produce the following code:

```
move ax, DGROUP : myA+2
move bx, [di+4]
```

This way, if you load a register (such as DI) with the address of a structure of type *myStruct*, you can use the member names to directly reference the members. The member name can be used in any position where a numeric constant is allowed in an assembly-statement operand.

The structure member must be preceded by a dot (.) to signal that a member name, rather than a normal C symbol, is being used. Member names are replaced in the assembly output by the numeric offset of the structure member (the numeric offset of *a_c* is 4), but no type information is retained. Thus members can be used as compile-time constants in assembly statements.

There is one restriction, however: if two structures that you're using in inline assembly have the same member name, you must distinguish between them. Insert the structure

type (in parentheses) between the dot and the member name, as if it were a cast. For example,

```
asm mov  bx,[di].(struct tm)tm_hour
```

Using jump instructions and labels

You can use any of the conditional and unconditional jump instructions, plus the loop instructions, in inline assembly. These instructions are valid only inside a function. Since no labels can be defined in the **asm** statements, jump instructions must use C **goto** labels as the object of the jump. If the label is too far away, the jump will not be automatically converted to a long-distance jump. For this reason, you should be careful when inserting conditional jumps. You can use the **-B** switch to check your jumps. Direct far jumps cannot be generated.

In the following code, the jump goes to the C **goto** label *a*.

```
int      x( )
{
a:                /* This is the goto label "a" */
    . . .
    asm jmp a      /* Goes to label "a" */
    . . .
}
```

Indirect jumps are also allowed. To use an indirect jump, use a register name as the operand of the jump instruction.

Compiling with inline assembly

There are two way Paradigm C++ can handle inline assembly code in your C or C++ code.

- Paradigm C++ can convert your C or C++ code into assembly language, then transfer to PASM to produce an .OBJ file. (This method is described in this section.)
- Paradigm C++ can use its built-in assembler (PASM) to insert your assembly statements directly into the compiler's instruction stream (16-bit compiler only). (This method is described in the following section.)

You can use the **-B** compiler option for inline assembly in your C or C++ program. If you can use this option, the compiler first generates an assembly file, then invokes PASM on that file to produce the .OBJ file.



By default, **-B** invokes PASM or PASM32. You can override it with **-Exxx**, where *xxx* is another assembler.

You can invoke PASM while omitting the **-B** option if you include the **#pragma inline** statement in your source code. This statement enables the **-B** option for you when the compiler encounters it. You will save compile time if you put **#pragma inline** at the top of your source file.

The **-B** option and **#pragma inline** tell the compiler to produce an .ASM file, which might contain your inline assembly instructions, and then transfer to PASM to assemble the .OBJ file. The 16-bit Paradigm C++ compiler has another method, PASM, that allows the compiler, not PASM, to assemble you inline assembly code.

Using the built-in assembler (PASM)

The 16-bit compiler can assemble your inline assembly instructions using the built-in assembler (PASM). This assembler is part of the compiler, and can do most of the things PASM can do, with the following restrictions:

- It can't use assembler macros.
- It can't handle 80386 or 80486 instruction.
- It doesn't permit Ideal mode syntax.
- It allows only a limited set of assembler directives (see page 10-213)

Because PASM isn't a complete assembler, it might not accept some assembly-language constructs. If this happens, Paradigm C++ will issue an error message. You then have two choices: you can simplify your inline assembly-language code so the assembler will accept it, or you can use the **-B** option to invoke PASM to catch whatever errors there might be. PASM might not identify the location of errors, however, because the original C source line number is lost.

Opcodes

You can include any of the 80x86 instruction opcodes as inline-assembly statements. There are four classes of instructions allowed by the Paradigm C++ compiler:

- Normal instructions - the regular 80x86 opcode set
- String instructions - special string-handling codes
- Jump instructions - various jump opcodes
- Assembly directives - data allocation and definition

All operands are allowed by the compiler, even if they are erroneous or disallowed by the assembler. The exact format of the operands is not enforced by the compiler.

Table 10-1 lists all allowable PASM opcodes. For 80286 instruction, use the **-2** command-line compiler option.



If you're using inline assembly in routines that use floating-point emulation (the command-line compiler option **-f**), the opcodes marked with * aren't supported.

Table 10-1
PASM opcode
mnemonics

PASM opcode mnemonics

aaa	fdivrp	fpatan	lsl
aad	feni	fprem	mov
aam	ffree*	fptan	mul
aas	fiadd	frndint	neg
adc	ficom	frstor	nop
add	ficom	fsave	not
and	fidiv	fscale	or
bound	fidivr	fsqrt	out
call	fild	fst	pop
cbw	fimul	fstcw	popa
clc	fincstp*	fstenv	popf
cld	finit	fstp	push
cli	fist	fstsw	pusha
cmc	fistp	fsub	pushf

cmp	fisub	fsubp	rcl
cwd	fisubr	fsubr	rcr
daa	fld	fsubrp	ret
das	fldl	ftst	rol
dec	fldcw	fwait	ror
div	fldenv	fxam	sahf
enter	fldl2e	fxch	sal
f2xm1	fldl2t	fxtract	sar
fabs	fldlg2	fyl2x	sbb
fadd	fldln2	fyl2xp1	shl
faddp	fldpi	hlt	shr
fbld	fldz	idiv	smsw
fbstp	fmul	imul	stc
fchs	fmulp	in	std
fclex	fnclex	inc	sti
fcom	fndisi	int	sub
fcom	fndisi	int	sub
fcomp	fneni	into	test
fcompp	fninit	iret	verr
fdecstp	fnop	lahf	verw
fdisi	fnsave	lds	wait
fdiv	fnstcw	lea	xchg
fdivp	fnstenv	leave	xlat
fdivr	fnstsw	les	xor

* Not supported if you're using inline assembly in routines that use floating-point emulation (the command-line compiler option **-f**).

When using 80186 instruction mnemonics in your inline-assembly statements, you must include the **-1** command-line option. This forces appropriate statements into the assembly-language compiler output so that the assembler will expect the mnemonics. If you're using an older assembler, these mnemonics might not be supported.

String instructions

In addition to the opcodes listed in Table 10-1, page 10-211, the string instructions given in Table 10-2 can be used alone or with repeat prefixes.

Table 10-2
PASM string instructions

PASM string instructions				
cmps	insw	movsb	outsw	stos
cmpsb	lods	movsw	scas	stosb
cmpsw	lodsb	scasb	stosw	
lodsw	outsb	scasw		
insb	movs			

The following prefixes can be used with the string instructions:

lock rep repe repnz repz

Jump instructions

Jump instructions are treated specially. Because a label can't be included on the instruction itself, jumps must go to C labels (see “Using jump instructions and labels,” page 10-210). The allowed jump instructions are given in the next table.

Table 10-3
Jump instructions

Jump instructions				
ja	jge	jnc	jns	loop
jae	jle	jne	jnz	loope
jb	jle	jng	jo	loopne
jbe	jmp	jnge	jp	loopnz
jc	jna	jnl	jpe	loopz
jcxz	jnae	jnl	jpo	
je	jnb	jno	js	
jg	jnbe	jnp	jz	

Assembly directives

The following assembly directives are allowed in Paradigm C++ inline-assembly statements:

db dd dw extrn

Header files summary

Header files, also called include files, provide function prototype declarations for library functions. Data types and symbolic constants used with the library functions are also defined in them, along with global variables defined by Paradigm C++ and by the library functions. The Paradigm C++ library follows the ANSI C standard on names of header files and their contents.



The middle column indicates C++ header files and header files defined by ANSI C.

alloc.h		Declares memory-management functions (allocation, deallocation, and so on).
assert.h	ANSI C	Defines the assert debugging macro.
bcd.h	C++	Declares the C++ class <i>bcd</i> and the overloaded operators for <i>bcd</i> and <i>bcd</i> math functions.
complex.h	C++	Declares the C++ complex math functions.
conio.h		Declares various functions used in calling the operating system console I/O routines.
constrea.h	C++	Defines the <i>conbuf</i> and <i>constream</i> classes.
ctype.h	ANSI C	Contains information used by the character classification and character conversion macros (such as <i>isalpha</i> and <i>toascii</i>).
date.h	C++	Defines the <i>date</i> class.
_defs.h		Defines the calling conventions for different application types and memory models.
dos.h		Defines various constants and gives declarations needed for DOS and 8086-specific calls
embedded.h		Defines various constants and gives declarations needed for embedded systems 8086-specific calls
errno.h	ANSI C	Defines constant mnemonics for the error codes.
except.h	C++	Declares the exception-handling classes and functions.
excpt.h		Declares C structured exception support.
fcntl.h		Defines symbolic constants used in connection with the library routine open.
float.h	ANSI C	Contains parameters for floating-point routines.
fstream.h	C++	Declares the C++ stream classes that support file input and output.
generic.h	C++	Contains macros for generic class declarations.
io.h		Contains structures and declarations for low-level input/output routines.
iomanip.h	C++	Declares the C++ streams I/O manipulators and contains templates for creating parameterized manipulators.
iostream.h	C++	Declares the basic C++ streams (I/O) routines.
limits.h	ANSI C	Contains environmental parameters, information about compile-time limitations, and ranges of integral quantities.
malloc.h		Declares memory-management functions and variables.
math.h	ANSI C	Declares prototypes for the math functions and math error handlers.

mem.h		Declares the memory-manipulation functions. (Many of these are also defined in string.h.)
memory.h		Contains memory-manipulation functions.
new.h	C++	Access to <i>_new_handler</i> , and <i>set_new_handler</i> .
_nfile.h		Defines the maximum number of open files.
_null.h		Defines the value of NULL.
process.h		Contains structures and declarations for terminating a program.
search.h		Declares functions for searching and sorting.
setjmp.h	ANSI C	Declares the functions <i>longjmp</i> and <i>setjmp</i> and defines a type <i>jmp_buf</i> that these functions use.
share.h		Defines parameters used in functions that make use of file-sharing.
signal.h	ANSI C	Defines constants and declarations for use by the <i>signal</i> and <i>raise</i> functions.
stdarg.h	ANSI C	Defines macros used for reading the argument list in functions declared to accept a variable number of arguments (such as <i>vprintf</i> , <i>vscanf</i> , and so on).
stddef.h	ANSI C	Defines several common data types and macros.
stdio.h	ANSI C	Defines types and macros needed for the standard I/O package defined in Kernighan and Ritchie and extended under UNIX System V. Defines the standard I/O predefined streams <i>stdin</i> , <i>stdout</i> , <i>stderr</i> , and <i>stderr</i> and declares stream-level I/O routines.
stdiost.h	C++	Declares the C++ stream classes for use with <i>stdio FILE</i> structures. You should use <i>iostream.h</i> for new code.
stdlib.h	ANSI C	Declares several commonly used routines such as conversion routines and search/sort routines.
string.h	ANSI C	Declares several string-manipulation and memory-manipulation routines.
strstrea.h	C++	Declares the C++ stream classes for use with byte arrays in memory.
sys\locking.h		Contains definitions for mode parameter of <i>locking</i> function.
sys\types.h		Declares the type <i>time_t</i> used with time functions.
time.h	ANSI C	Defines a structure filled in by the time-conversion routines <i>asctime</i> , <i>localtime</i> , and <i>gmtime</i> , and a type used by the routines <i>ctime</i> , <i>difftime</i> , <i>gmtime</i> , <i>localtime</i> , and <i>stime</i> . It also provides prototypes for these routines.
typeinfo.h	C++	Declares the run-time type information classes.
values.h		Defines important constants, including machine dependencies; provided for UNIX System V compatibility.
varargs.h		Definitions for accessing parameters in functions that accept a variable number of arguments. Provided for UNIX compatibility; you should use <i>stdarg.h</i> for new code.

Using precompiled headers

Paradigm C++ can generate (and subsequently use) precompiled headers to speed up your project compile times.

Precompiled headers are header files that are compiled once, then used over and over again in their compiled state.

You can use a precompiled header if a compilation uses one or more of the same header files, the same compiler options, the same macro defines, and so on, as is contained in the precompiled header file.

To control the use of precompiled headers, do one of the following:

- From within the IDE, turn on the Precompiled Headers option in the Compiler settings page of the Project Options dialog box. The IDE bases the name of the precompiled header file on the project name, creating `<PROJECT_NAME>.CSM`.
- From the command line, use the following command-line options:
-H=<filename>, **-Hc**, **-H<filename>**, and **-Hu**.
- From within your code, use the **hdrfile** and **hdrstop** pragmas.

Setting file names

Paradigm C++ stores all precompiled headers in one file, using the following naming convention:

- The 16-bit command-line compiler names the precompiled header file `PCDEF.CSM`
- The 32-bit command-line compiler names the precompiled header file `PC32DEF.CSM`
- The IDE names the precompiled header file `<PROJECT_NAME>.CSM`.



To explicitly set the precompiled file name from the command line, use the **-H=<filename>** option or the `#pragma hdrfile` directive.

Precompiled header file overview

When compiling C and C++ programs, the compiler can spend up to half its time parsing header files. When the compiler parses a header file, it enters declarations and definitions into its symbol table.

Precompiled headers cut this process short by creating and storing a binary image of the symbol table on disk. By directly loading a binary image of the symbol table, the compiler can increase the speed of this step by over ten times. The disadvantage is that precompiled header files can become quite large because they can contain the symbol table images for all the **#include** files encountered in your sources.

If, while compiling a source file, Paradigm C++ discovers that the first **#include** files are identical to those of a previous compilation (of either the same or different source), it loads the binary image for those **#include** files and parses the remaining **#include** files.

For a given module, either all or none of the precompiled headers are used--if compilation of any included header file fails, the precompiled header file isn't updated for that module.

Precompiled header limits

When using precompiled headers, `PCDEF.CSM` can become very large because it contains symbol table images for all sets of includes encountered in your sources. If you don't have sufficient disk space, you'll get a warning saying the write failed because of the precompiled headers. To fix this, you must provide more disk space and retry the compile. For information on reducing the size of the `PCDEF.CSM` file, see "Optimizing precompiled headers," page 11-218.

If you're using large macros in a makefile in addition to using precompiled headers, there is a limit on the macro size: 4K for 16-bit applications and 16K for 32-bit applications.

If a header file contains any code, it can't be precompiled. For example, although C++ class definitions can appear in header files, you should ensure that only inline member

functions are defined in the header and heed warnings such as Functions containing reserved word are not expanded inline.

Precompiled header rules

The following rules apply when you create and use precompiled headers:

1. A header that contains code can't be precompiled. For example, although C++ class definitions can appear in header files, make sure that only inline member functions are defined in the header. Heed warnings such as Functions containing 'for' are not expanded inline.
2. In order to use a previously generated precompiled header, the source file must:
 - Have the same set of include files, in the same order, as the precompiled header
 - Have the same macros defined with identical values as the precompiled header
 - Use the same language (C or C++) as the precompiled header
 - Use header files with identical time stamps as the precompiled header
3. In addition, the following option settings must be identical to those used when you generated the precompiled header:
 - Memory model, including SS != DS (**-mx**)
 - Underscores on externs (**-u**)
 - Maximum identifier length (**-iL**)
 - Target DOS or Windows (**-W** or **-Wx**)
 - Generate word alignment (**-a**)
 - Pascal calls (**-p**)
 - Treat enums as integers (**-b**)
 - Default char is unsigned (**-K**)
 - Virtual table control (**-Vx** and **-Vmx**)
 - Expand intrinsic functions inline (**-Oi**)
 - Templates (**-Jx**)
 - String literals in code segment (**-dc**, 16-bit)
 - Debugging information (**-v**, **-vi**, and **-R**)
 - Far variables (**-Fx**)
 - Language compliance (**-A**)
 - C++ compile (**-P**)
 - DOS overlay-compatible code (**-Y**)
4. If you're using large macros in addition to using precompiled headers, the compiler limits the size of the macros as following:
 - 4K macros for 16-bit applications
 - 16K macros for 32-bit applications

Optimizing precompiled headers

For the most efficiently compiled precompiled headers, follow these rules:

- Arrange your header files in the same sequence in all source files.
- Put the largest header files first.
- Prime the precompiled header file with often-used initial sequences of header files.

- Use `#pragma hdrstop` to terminate the list of header files at well-chosen places. This lets you make the list of header files in different sources look similar to the compiler.

For example, suppose you have the following two source files (A_SOURCE.CPP and B_SOURCE.CPP), which both include windows.h and myhdr.h:

```
/* A_SOURCE.CPP */
#include <windows.h>
#include "myhdr.h"
#include "xxx.h"
// ...

/* B_SOURCE.CPP */
#include "yyy.h"
#include <string.h>
#include "myhdr.h"
#include <windows.h>
// ...
```

To optimize the precompiled headers for these source files, you would rearrange the beginning of B_SOURCE.CPP as follows:

```
/* Revised B_SOURCE.CPP */
#include <windows.h>
#include "myhdr.h"
#include "yyy.h"
#include <string.h>
// ...
```

Now, windows.h and myhdr.h are in the same order in both A_SOURCE.CPP and B_SOURCE.CPP, and they are both located at the beginning of the `#include` list.

In addition, you could also create a new source file called PREFIX.CPP which contains only the matching header files, like this:

```
/* PREFIX.CPP */
#include <windows.h>
#include "myhdr.h"
```

If you compile PREFIX.CPP first (or insert a `#pragma hdrstop` in both A_SOURCE.CPP and B_SOURCE.CPP), the net effect is that after the initial compilation of PREFIX.CPP, both A_SOURCE.CPP and B_SOURCE.CPP will be able to load the symbol table produced by PREFIX.CPP. The compiler will then need to parse only xxx.h for A_SOURCE.CPP, and yyy.h and strings.h for B_SOURCE.CPP.

alloc.h

Declares memory-management functions (allocation, deallocation, and so on).

Functions

- calloc
- farcalloc
- farfree
- farmalloc
- farrealloc
- free
- heapcheck
- heapcheckfree

- heapchecknode
- heapfillfree
- heapwalk
- malloc
- realloc

Constants, data types and global variables

- NULL
- ptrdiff_t
- size_t

assert.h

Defines the assert debugging macro.

Functions

- assert

ctype.h

Contains information used by the character classification and character conversion macros.

Functions and macros

- isalnum
- isalpha
- isascii
- iscntrl
- isdigit
- isgraph
- islower
- isprint
- ispunct
- isspace
- isupper
- isxdigit
- toascii
- _tolower
- tolower
- _toupper
- toupper

Constants, data types and global variables

- _IS_CTL
- _IS_DIG
- _IS_HEX
- _IS_LOW

- `_IS_PUN`
- `_IS_SP`
- `_IS_UPP`

dos.h

Defines various constants and gives declarations needed for DOS and 8086-specific calls.

Functions and macros

- `_chain_intr`
- `disable`
- `_emit_`
- `enable`
- `FP_OFF`
- `FP_SEG`
- `getvect`
- `inport`
- `inportb`
- `int86`
- `in86x`
- `intr`
- `MK_FP`
- `outport`
- `outportb`
- `peek`
- `peekb`
- `poke`
- `pokeb`
- `segread`
- `setvect`

Constants, data types and global variables

- `errno`
- `SREGS`

embedded.h

Defines various constants and gives declarations needed for embedded systems 8086-specific calls.

Functions and macros

- `_chain_intr`
- `disable`
- `_emit_`
- `enable`

- FP_OFF
- FP_SEG
- getvect
- inport
- inportb
- int86
- int86x
- intr
- MK_FP
- outport
- outportb
- peek
- peekb
- poke
- pokeb
- segread
- setvect

Constants, data types and global variables

- SREGS

errno.h

Defines constant mnemonics for the error codes.

Constants, data types and global variables

- _doserrno
- errno
- _sys_errlist
- _sys_nerr
- error number definitions

fcntl.h

Defines open flags for open and similar library functions.

Functions

- _fmode

Constants

- O_APPEND
- O_BINARY
- O_CHANGED
- O_CREAT
- O_DENYALL
- O_DENYNONE
- O_DENYREAD

- O_DENYWRITE
- O_DEVICE
- O_EXCL
- O_NOINHERIT
- O_RDONLY
- O_RDWR
- O_TEXT
- O_TRUNC
- O_WRONLY

float.h

Contains parameters for floating-point routines.

Functions

- _clear87
- _fpreset
- _status87

Constants, data types and global variables

- CW_DEFAULT
- FPE_EXPLICITGEN
- FPE_INEXACT
- FPE_INTDIV0
- FPE_INTOVFLOW
- FPE_INVALID
- FPE_OVERFLOW
- FPE_UNDERFLOW
- FPE_ZERODIVIDE
- ILL_EXECUTION
- ILL_EXPLICITGEN
- SEGV_BOUND
- SEGV_EXPLICITGEN

generic.h

Contains macros for generic class declarations.

io.h

Contains structures and declarations for low-level input/output routines.

Functions

- setmode

Constants, data types and global variables

- HANDLE_MAX

iomanip.h

Declares the C++ streams I/O manipulators and contains macros for creating parameterized manipulators.

Includes

- `iostream.h`

Classes

- `iapply`
- `imanip`
- `ioapp`
- `iomanip`
- `oapp`
- `omanip`
- `sapp`
- `smanip`

Overloaded Operators

`<<` `>>`

limits.h

Contains environmental parameters, information about compile-time limitations, and ranges of integral quantities.

Constants, data types and global variables

- `CHAR_BIT`
- `CHAR_MAX`
- `CHAR_MIN`
- `INT_MAX`
- `INT_MIN`
- `LONG_MAX`
- `LONG_MIN`
- `SCHAR_MAX`
- `SCHAR_MIN`
- `SHRT_MAX`
- `SHRT_MIN`
- `UCHAR_MAX`
- `UINT_MAX`
- `ULONG_MAX`
- `USHRT_MAX`

malloc.h

Declares memory-management functions and variables.

Includes

- ALLOC.H

Functions

- _heapchk
- _heapmin
- _heapset
- stackavail

math.h

Declares prototypes for the math functions and math error handlers.

Functions

- abs
- acos, acosl
- asin, asinl
- atan, atanl
- atan2, atan2l
- atof, _atold
- cabs, cabsl
- ceil, ceill
- cos, cosl
- cosh, coshl
- exp, expl
- fabs, fabs
- floor, floorl
- fmod, fmodl
- frexp, frexpl
- hypot, hypotl
- labs
- ldexp, ldexpl
- log, logl
- log10, log10l
- _matherr, _matherrl
- modf, modfl
- poly, poly1
- pow, powl
- pow10, pow10l
- sin, sinl
- sinh, sinhl
- sqrt, sqrtl
- tan, tanl
- tanh, tanhl

Constants, data types and global variables

- `complex` (struct)
- `_complexl` (struct)
- `EDOM`
- `ERANGE`
- `exception` (struct)
- `_exceptionl` (struct)
- `HUGE_VAL`
- `M_E`
- `M_LOG2E`
- `M_LOG10E`
- `M_LN2`
- `M_LN10`
- `M_PI`
- `M_PI_2`
- `M_PI_4`
- `M_1_PI`
- `M_2_PI`
- `M_1_SQRTPI`
- `M_2_SQRTPI`
- `M_SQRT2`
- `M_SQRT_2`
- `_mexcep`

mem.h

Declares the memory-manipulation functions. (Many of these are also defined in `string.h`.)

Functions

- `_fmemccpy`
- `_fmemchr`
- `_fmemcmp`
- `_fmemcpy`
- `_fmemicmp`
- `_fmemmove`
- `_fmemset`
- `_fmovmem`
- `memccpy`
- `memchr`
- `memcmp`
- `memcpy`
- `memicmp`
- `memmove`

- `memset`
- `movedata`
- `movmem`
- `setmem`

Constants, Data Types and Global Variables

- `NULL`
- `ptrdiff_t`
- `size_t`

memory.h

Contains memory-manipulation functions.

Includes

- `MEM.H`

new.h

Provides access to the the following functions:

- `set_new_handler`
- `_new_handler` (global variable)

process.h

Contains structures and declarations for terminating a program.

Functions

- `abort`
- `_c_exit`
- `_cexit`
- `exit`
- `_exit`

search.h

Declares functions for searching and sorting.

Functions

- `bsearch`
- `lfind`
- `lsearch`
- `qsort`

setjmp.h

Declares the functions `longjmp` and `setjmp` and defines a type `jmp_buf` that these functions use.

Functions

- longjmp
- setjmp

Constants, data types and global variables

- jmp_buf

share.h

Defines parameters used in functions that make use of file-sharing.

Constants, data types and global variables

- SH_COMPAT
- SH_DENYNO
- SH_DENYNONE
- SH_DENYRD
- SH_DENYRW
- SH_DENYWR

signal.h

Defines constants and declarations for use by the signal and raise functions.

Functions

- raise
- signal

Constants, data types and global variables

- predefined signal handlers
- sig_atomic_t type
- SIG_DFL
- SIG_ERR
- SIG_IGN
- SIGABRT
- SIGFPE
- SIGILL
- SIGINT
- SIGSEGV
- SIGTERM

stdarg.h

Defines macros used for reading the argument list in functions declared to accept a variable number of arguments (such as vprintf, vscaf, and so on).

Macros

- va_arg
- va_end

- va_start

Constants, data types and global variables

- va_list

stddef.h

Defines several common data types and macros.

Functions

- offsetof

Constants, data types and global variables

- NULL
- ptrdiff_t
- size_t
- wchar_t

stdio.h

Defines types and macros needed for the standard I/O package defined in Kernighan and Ritchie and extended under UNIX System V. It defines the standard I/O predefined streams stdin, stdout, stderr, and declares stream-level I/O routines.

Functions

_fstrncpy	setbuf
getc	setvbuf
getchar	sprintf
gets	sscanf
getw	_strerror
perror	strerror
printf	strncpy
putc	ungetc
putchar	vprintf
puts	vscanf
putw	vsprintf
scanf	vsscanf

Constants, data types and global variables

_F_BIN	FILE	size_t
_F_BUF	FOPEN_MAX	stdaux
_F_EOF	fpos_t	stderr
_F_ERR	_IOFBF	stdin
_F_IN	_IOLBF	stdout
_F_LBUF	_IONBF	stdprn
_F_OUT	L_ctermid	SYS_OPEN
_F_RDWR	NULL	TMP_MAX

stdiostr.h

Declares the C++ stream classes for use with stdio FILE structures. You should use `iostream.h` for new code.

Includes

- `IOSTREAM.H`
- `STDIO.H`

stdlib.h

Declares several commonly used routines such as conversion routines and search/sort routines.

Functions

<code>abort</code>	<code>labs</code>	<code>realloc</code>
<code>abs</code>	<code>ldiv</code>	<code>_rotl</code>
<code>atexit</code>	<code>lfind</code>	<code>_rotr</code>
<code>atof</code>	<code>_lrotl</code>	<code>srand</code>
<code>atoi</code>	<code>_lrotr</code>	<code>strtod</code>
<code>atol</code>	<code>lsearch</code>	<code>strtol</code>
<code>bsearch</code>	<code>ltoa</code>	<code>_strtold</code>
<code>calloc</code>	<code>malloc</code>	<code>strtoul</code>
<code>_crotr</code>	<code>max</code>	<code>swab</code>
<code>div</code>	<code>mblen</code>	<code>ultoa</code>
<code>ecvt</code>	<code>mbstowcs</code>	<code>wcstombs</code>
<code>exit</code>	<code>mbtowc</code>	<code>wctomb</code>
<code>_exit</code>	<code>min</code>	
<code>fcvt</code>	<code>qsort</code>	
<code>free</code>	<code>rand</code>	
<code>gcvt</code>	<code>random</code>	
<code>itoa</code>	<code>randomize</code>	

Constants, data types and global variables

- `div_t`
- `_doserrno`
- `errno`
- `EXIT_FAILURE`
- `EXIT_SUCCESS`
- `_fmode`
- `ldiv_t`
- `NULL`
- `RAND_MAX`
- `size_t`
- `sys_errlist`

- sys_nerr
- wchar_t

string.h

Declares several string-manipulation and memory-manipulation routines.

Functions

_fmemccpy	_fstrset	strdup
_fmemchr	_fstrspn	strdup
_fmemcmp	_fstrstr	strerror
_fmemcpy	_fstrtok	_strerror
_fmemicmp	_fstrupr	stricmp
_fmemset	memcpy	strlen
_fstr*	memchr	strlwr
_fstrcat	memcmp	strncat
_fstrchr	memcpy	strncmp
_fstrcmp	memicmp	strncmpi
_fstrcpy	memmove	strncpy
_fstrcspn	memset	strnicmp
_fstrdup	movedata	strnset
_fstricmp	movmem	strpbrk
_fstrlen	setmem	strchr
_fstrlwr	strcpy	strev
_fstrncat	strcat	strset
_fstrncmp	strchr	strspn
_fstrncpy	strcmp	strstr
_fstrnicmp	strcmp	strtok
_fstrnset	strcmpi	strupr
_fstrpbrk	strcoll	strxfrm
_fstrchr	strcpy	
_fstrrev	strcspn	

Constants, data types and global variables

- size_t

sys\locking.h

Contains definitions for mode parameter of locking function.

sys\types.h

Constants, data types and global variables

- time_t

time.h

Defines a structure filled in by time-conversion routines `asctime`, `localtime`, and `gmtime`, and a type used by the routines `ctime`, `difftime`, `gmtime`, `localtime` and `stime`. It also provides prototypes for these routines.

Functions

- `asctime`
- `ctime`
- `difftime`
- `gmtime`
- `localtime`
- `mktime`
- `randomize`
- `stime`
- `_strdate`
- `strftime`
- `_strtime`
- `time`

Constants, Data Types and Global Variables

- `size_t`
- `time_t`
- `tm`

values.h

Defines UNIX compatible constants for limits to float and double values.

Functions

- `BITSPERBYTE`
- `DMAXEXP`
- `DMAXPOWTWO`
- `DMINEXP`
- `DSIGNIF`
- `FMAXEXP`
- `FMAXPOWTWO`
- `FMINEXP`
- `FSIGNIF`
- `_FEXPLEN`
- `HIBITI`
- `HIBITL`
- `HIBITS`
- `_LENBASE`
- `MAXDOUBLE`
- `MAXFLOAT`

- MAXINT
- MAXLONG
- MAXSHORT
- MINDOUBLE
- MINFLOAT

varargs.h

Definitions for accessing parameters in functions that accept a variable number of arguments.

These macros are compatible with UNIX System V.

Use STDARG.H for ANSI C compatibility.



You can't include both STDARG.H and VARARGS.H

Macros

- va_start
- va_arg
- va_end

Type

- va_list

excpt.h

The excpt.h header file contains the declarations and prototypes for structured exception-handling values, types, and routines.

_defs.h

The _defs.h header file contains common definitions for pointer size and calling conventions.

Calling Conventions

_RTLENTY	Specifies the calling convention used by the Standard Run-time Library.
_USERENTRY	Specifies the calling convention the Standard Run-time Library expects user-compiled functions to use for callbacks.

Export (and size for DOS) information

_EXPCCLASS	Exports the class if you are building a DLL version of a library.
_EXPDATA	Exports the data if you are building a DLL version of a library.
_EXPFUNC	Exports the function if you are building a DLL version of a library.



These export macros are provided as examples only and should not be used to create user-defined functions.

_nfile.h

The _nfile.h header file defines _NFILE_, which specifies the maximum number of open files you can have.

NFILE is defined as 50 for all applications.

_null.h

The `_null.h` defines the value of NULL for different memory models and applications types:

Model	Value	
Flat	((void *)0)	if not C++ or Windows application
Flat	0	
Small	0	
Medium	0	
Large	0L	

This chapter describes the floating-point options and explains how to use *complex* and *bcd* numerical types.

Floating-point I/O

Floating-point output requires linking of conversion routines used by *printf*, *scanf*, and any variants of these functions. To reduce executable size, the floating-point formats are not automatically linked. However, this linkage is done automatically whenever your program uses a mathematical routine or the address is taken of some floating-point number. If neither of these actions occur, the missing floating-point formats can result in a run-time error.

The following program illustrates how to set up your program to properly execute.

```
/* PREPARE TO OUTPUT FLOATING-POINT NUMBERS. */
#include <stdio.h>

#pragma extref _floatconvert

void main() {
    printf("d = %f\n", 1.3);
}
```

Floating-point options

There are two types of numbers you work with in C: integer (**int**, **short**, **long**, and so on) and floating point (**float**, **double**, and **long double**). Your computer's processor can easily handle integer values, but more time and effort are required to handle floating-point values.

However, the iAPx86 family of processors has a corresponding family of math coprocessors, the 8087, the 80287, and the 80387. We refer to this entire family of math coprocessors as the 80x87, or "the coprocessor."

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly. If you use floating point a lot, you'll probably want a coprocessor. The CPU in your computer interfaces to the 80x87 via special hardware lines.



If you have an 80486 or Pentium processor, the numeric coprocessor is probably already built in.

Emulating the 80x87 chip

The default Paradigm C++ code-generation option is *emulation* (the **-f** command-line compiler option). This option is for programs that might or might not have floating point, and for machines that might or might not have an 80x87 math coprocessor.

With the emulation option, the compiler will generate code as if the 80x87 were present, but will also link in the emulation library (EMU.LIB). When the program runs, it uses the 80x87 if it is present; if no coprocessor is present at run-time, it uses special software that emulates the 80x87. This software uses 512 bytes of your stack, so make allowance for it when using the emulation option and set your stack size accordingly.

Using the 80x87 code

If your program is going to run only on machines that have an 80x87 math coprocessor, you can save a small amount in your .EXE file size by omitting the 80x87 autodetection and emulation logic. Choose the 80x87 floating-point code-generation option (the **-f87** command-line compiler option). Paradigm C++ will then link your programs with FP87.LIB instead of with EMU.LIB.

No floating-point code

If there is no floating-point code in your program, you can save a small amount of link time by choosing None for the floating-point code-generation option (the **-f-** command-line compiler option). Then Paradigm C++ will not link with EMU.LIB, FP87.LIB, or MATHx.LIB.

Fast floating-point option

Paradigm C++ has a fast floating-point option (the **-ff** command-line compiler option). It can be turned off with **-ff-** on the command line. Its purpose is to allow certain optimizations that are technically contrary to correct C semantics. For example,

```
double x;  
x = (float)(3.5*x);
```

To execute this correctly, x is multiplied by 3.5 to give a **double** that is truncated to **float** precision, then stored as a **double** in x . Under the fast floating-point option, the **long double** product is converted directly to a **double**. Since very few programs depend on the loss of precision in passing to a narrower floating-point type, fast floating point is the default.

The 87 environment variable

If you build your program with 80x87 emulation, which is the default, your program will automatically check to see if an 80x87 is available, and will use it if it is.

There are some situations in which you might want to override this default autodetection behavior. For example, your own run-time system might have an 80x87, but you might need to verify that your program will work as intended on systems without a coprocessor. Or your program might need to run on a PC-compatible system, but that particular system returns incorrect information to the autodetection logic (saying that a nonexistent 80x87 is available, or vice versa).

Paradigm C++ provides an option for overriding the start-up code's default autodetection logic; this option is the 87 environment variable.

You set the 87 environment variable at the DOS prompt with the SET command, like this:

```
C> SET 87=N
```

or like this:

```
C> SET 87=Y
```

Don't include spaces on either side of the =. Setting the 87 environment variable to N (for No) tells the start-up code that you do not want to use the 80x87, even though it might be present in the system.



Setting the 87 environment variable to Y (for Yes) means that the coprocessor is there, and you want the program to use it. *Let the programmer beware:* If you set `87 = Y` when, in fact, there is no 80x87 available on that system, your system will hang.

If the 87 environment variable has been defined (to any value) but you want to undefine it, enter the following at the DOS prompt:

```
C> SET 87=
```

Press *Enter* immediately after typing the equal sign.

Registers and the 80x87

When you use floating point, make note of these points about registers:

- In 80x87 emulation mode, register wrap-around and certain other 80x87 peculiarities are not supported.
- If you are mixing floating point with inline assembly, you might need to take special care when using 80x87 registers. Unless you are sure that enough free registers exist, you might need to save and pop the 80x87 registers before calling functions that use the coprocessor.

Disabling floating-point exceptions

By default, Paradigm C++ programs abort if a floating-point overflow or divide-by-zero error occurs. You can mask these floating-point exceptions by a call to `_control87` in *main*, before any floating-point operations are performed. For example,

```
#include <float.h>
main() {
    _control87(MCW_EM,MCW_EM);
    ...
}
```



You can determine whether a floating-point exception occurred after the fact by calling `_status87` or `_clear87`. See "Run-time library functions" in the online Help index for details about these functions.

Certain math errors can also occur in library functions; for instance, if you try to take the square root of a negative number. The default behavior is to print an error message to the screen, and to return a NAN (an IEEE not-a-number). Use of the NAN is likely to cause a floating-point exception later, which will abort the program if unmasked. If you don't want the message to be printed, insert the following version of `_matherr` into your program:

```
#include <math.h>
int _matherr(struct _exception *e)
{
    return 1;                /* error has been handled */
}
```

Any other use of `_matherr` to intercept math errors is not encouraged; it is considered obsolete and might not be supported in future versions of Paradigm C++.

Using complex types

Complex numbers are numbers of the form $x + yi$, where x and y are real numbers, and i is the square root of -1 . Paradigm C++ as always had a type:

```
struct complex
{
    double  x, y;
};
```

defined in `math.h`. This type is convenient for holding complex numbers, because they can be considered a pair of real numbers. However, the limitations of C make arithmetic with complex numbers rather cumbersome. With the addition of C++, complex math is much simpler.

A significant advantage to using the Paradigm C++ *complex* numerical type is that all of the ANSI C Standard mathematical routines are defined to operate with it. These mathematical routines are not defined for use with the C **struct complex**.



See "complex class" in the online Help Book Shelf index for more information. The Book Shelf index can be accessed by choosing Help|Keyboard and clicking on the Book Shelf menu tab.

To use complex numbers in C++, all you have to do is to include `complex.h`. In `complex.h`, all the following have been overloaded to handle complex numbers:

- All of the binary arithmetic operators.
- the input and output operators, `>>` and `<<`.
- the ANSI C math functions.

The complex library is invoked only if the argument is of type *complex*. Thus, to get the complex square root of -1 , use

```
sqrt(complex(-1))
```

and not

```
sqrt(-1)
```

The following functions are defined by class `complex`:

```
double  arg(complex&);      // angle in the plane
complex conj(complex&);    // complex conjugate
double  imag(complex&);    // imaginary part
double  norm(complex&);    // square of the magnitude
double  real(complex&);    // real part
// Use polar coordinates to create a complex.
complex polar(double mag, double angle = 0);
```

Using bcd types

Paradigm C++, along with almost every other computer and compiler, does arithmetic on binary numbers (that is, base 2). This can sometimes be confusing to people who are used to decimal (base 10) representations. Many numbers that are exactly representable in base 10, such as 0.01, can only be approximated in base 2.



See "bcd class" in the online Help Book Shelf index for more information. The Book Shelf index can be accessed by choosing Help|Keyboard and clicking on the Book Shelf menu tab.

Binary numbers are preferable for most applications, but in some situations the round-off error involved in converting between base 2 and 10 is undesirable. The most

common example of this is a financial or accounting application, where the pennies are supposed to add up. Consider the following program to add up 100 pennies and subtract a dollar:

```
#include <stdio.h>
int i;
float x = 0.0;
for (i = 0; i < 100; ++i)
    x += 0.01;
x -= 1.0;
printf("100*.01 - 1 = %g\n",x);
```

The correct answer is 0.0, but the computed answer is a small number close to 0.0. The computation magnifies the tiny round-off error that occurs when converting 0.01 to base 2. Changing the type of *x* to **double** or **long double** reduces the error, but does not eliminate it.

To solve this problem, Paradigm C++ offers the C++ type *bcd*, which is declared in *bcd.h*. With *bcd*, the number 0.01 is represented exactly, and the *bcd* variable *x* provides an exact penny count.

```
#include <bcd.h>
int i;
bcd x = 0.0;
for (i = 0; i < 100; ++i)
    x += 0.01;
x -= 1.0;
cout << "100*.01 - 1 = " << x << "\n";
```

Here are some facts to keep in mind about *bcd*:

- *bcd* does not eliminate all round-off error: A computation like $1.0/3.0$ will still have round-off error.
- *bcd* types can be used with ANSI C math functions.
- *bcd* numbers have about 17 decimal digits precision, and a range of about 1×10^{-125} to 1×10^{125} .

Converting bcd numbers

bcd is a defined type distinct from **float**, **double**, or **long double**; decimal arithmetic is performed only when at least one operand is of the type *bcd*.

The *bcd* member function *real* is available for converting a *bcd* number back to one of the usual formats (**float**, **double**, or **long double**), though the conversion is not done automatically. *real* does the necessary conversion to **long double**, which can then be converted to other types using the usual C conversions. For example, a *bcd* can be printed using any of the following four output statements with *cout* and *printf*.

```

/* PRINTING bcd NUMBERS */
/* This must be compiled as a C++ program. */
#include <bcd.h>
#include <iostream.h>
#include <stdio.h>

void main(void) {
    bcd a = 12.1;
    double x = real(a); // This conversion required for printf().

    printf("\na = %g", x);
    printf("\na = %Lg", real(a));
    printf("\na = %g", (double)real(a));
    cout << "\na = " << a; // the preferred method.
}

```



Since *printf* doesn't do argument checking, the format specifier must have the L if the long double value *real(a)* is passed.

Number of decimal digits

You can specify how many decimal digits after the decimal point are to be carried in a conversion from a binary type to a *bcd*. The number of places is an optional second argument to the constructor *bcd*. For example, to convert \$1000.00/7 to a *bcd* variable rounded to the nearest penny, use

```
bcd a = bcd(1000.00/7, 2)
```

where 2 indicates two digits following the decimal point. Thus,

1000.00/7	=	142.85714...
bcd(1000.00/7, 2)	=	142.860
bcd(1000.00/7, 1)	=	142.900
bcd(1000.00/7, 0)	=	143.000
bcd(1000.00/7, -1)	=	140.000
bcd(1000.00/7, -2)	=	100.000

The number is rounded using banker's rounding (as specified by IEEE), which rounds to the nearest whole number, with ties being rounded to an even digit. For example,

bcd(12.335, 2)	=	12.34
bcd(12.345, 2)	=	12.34
bcd(12.355, 2)	=	12.36

16-bit memory management

This chapter discusses

- What to do when you receive "Out of memory" errors.
- What memory models are: how to choose one, and why you would (or wouldn't) want to use a particular memory model.

Running out of memory

Paradigm C++ does not generate any intermediate data structures to disk when it is compiling (Paradigm C++ writes only .OBJ files to disk); instead it uses RAM for intermediate data structures between passes. Because of this, you might encounter the message "Out of memory" if there isn't enough memory available for the compiler.

The solution to this problem is to make your functions smaller, or to split up the file that has large functions.

Memory models

Paradigm C++ gives you five memory models, each suited for different program and code sizes. Each memory model uses memory differently. What do you need to know to use memory models? To answer that question, you need to take a look at the computer system you're working on. Its central processing unit (CPU) is a microprocessor belonging to the Intel iAPx86 family; an 80286, 80386, 80486, or Pentium. For now, we'll just refer to it as an 8086.



See page 13-247 for a summary of each memory model.

The 8086 registers

The following figure shows some of the registers found in the 8086 processor. There are other registers—because they can't be accessed directly, they aren't shown here.

Figure 13-1
8086 registers

General-purpose registers	
AX	accumulator (math operations) AH AL
BX	base (indexing) BH BL
CX	count (indexing) CH CL
DX	data (holding data) DH DL
Segment address registers	
CS	code segment pointer
DS	data segment pointer
SS	stack segment pointer
ES	extra segment pointer
Special-purpose registers	
SP	stack pointer
BP	base pointer
SI	source index
DI	destination index

General-purpose registers

The general-purpose registers are the registers used most often to hold and manipulate data. Each has some special functions that only it can do. For example,

- Some math operations can only be done using AX.
- BX can be used as an index register.
- CX is used by LOOP and some string instructions.
- DX is implicitly used for some math operations.

But there are many operations that all these registers can do; in many cases, you can freely exchange one for another.

Segment registers

The segment registers hold the starting address of each of the four segments. As described in the next section, the 16-bit value in a segment register is shifted left 4 bits (multiplied by 16) to get the true 20-bit address of that segment.

Special-purpose registers

The 8086 also has some special-purpose registers:

- The SI and DI registers can do many of the things the general-purpose registers can, plus they are used as index registers. They're also used by Paradigm C++ for register variables.
- The SP register points to the current top-of-stack and is an offset into the stack segment.
- The BP register is a secondary stack pointer, usually used to index into the stack in order to retrieve arguments or automatic variables.

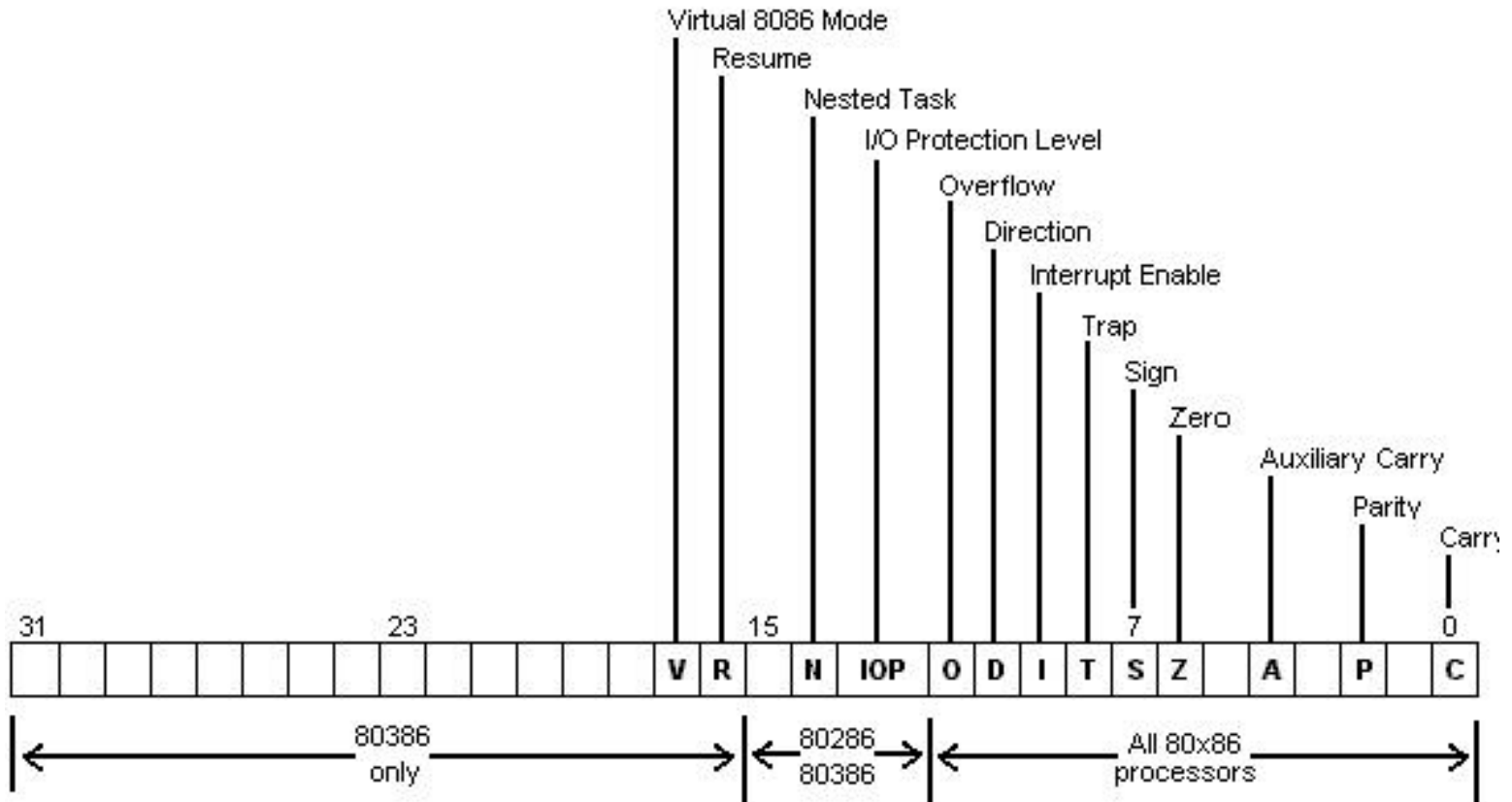
Paradigm C++ functions use the base pointer (BP) register as a base address for arguments and automatic variables. Parameters have positive offsets from BP, which vary depending on the memory model. BP points to the saved previous BP value if there is a stack frame. Functions that have no arguments will not use or save BP if the Standard Stack Frame option is *Off*.

Automatic variables are given negative offsets from BP. The offsets depend on how much space has already been assigned to local variables.

The flags register

The 16-bit flags register contains all pertinent information about the state of the 8086 and the results of recent instructions.

Figure 13-2 Flags register of 80x86 processors



For example, if you wanted to know whether a subtraction produced a zero result, you would check the *zero flag* (the Z bit in the flags register) immediately after the instruction; if it were set, you would know the result was zero. Other flags, such as the carry and *overflow flags*, similarly report the results of arithmetic and logical operations.

Other flags control the 8086 operation modes. The *direction flag* controls the direction in which the string instructions move, and the *interrupt flag* controls whether external hardware, such as a keyboard or modem, is allowed to halt the current code temporarily so that urgent needs can be serviced. The *trap flag* is used only by software that debugs other software.

The flags register isn't usually modified or read directly. Instead, the flags register is generally controlled through special assembler instructions (such as **CLD**, **STI**, and **CMC**) and through arithmetic and logical instructions that modify certain flags. Likewise, the contents of certain bits of the flags register affect the operation of instructions such as **JZ**, **RCR**, and **MOVS**. The flags register is not really used as a storage location, but rather holds the status and control data for the 8086.

Memory segmentation

The Intel 8086 microprocessor has a *segmented memory architecture*. It has a total address space of 1 MB, but is designed to directly address only 64K of memory at a time. A 64K chunk of memory is known as a segment; hence the phrase "segmented memory architecture."

- The 8086 keeps track of four different segments: *code*, *data*, *stack*, and *extra*. The code segment is where the machine instructions are; the data segment is where information is; the stack is, of course, the stack; and the extra segment is also used for extra data.
- The 8086 has four 16-bit segment registers (one for each segment) named CS, DS, SS, and ES; these point to the code, data, stack, and extra segments, respectively.
- A segment can be located anywhere in memory. In DOS real-mode it can be located almost anywhere. For reasons that will become clear as you read on, a segment must start on an address that is evenly divisible by 16 (in decimal).

Address calculation

A complete address on the 8086 is composed of two 16-bit values: the segment address and the offset. Suppose the data segment address—the value in the DS register—is 2F84 (base 16), and you want to calculate the actual address of some data that has an offset of 0532 (base 16) from the start of the data segment: how is that done?

Address calculation is done as follows: Shift the value of the segment register 4 bits to the left (equivalent to one hex digit), then add in the offset.

The resulting 20-bit value is the actual address of the data, as illustrated here:

DS register (shifted):	0010 1111 1000 0100 0000	=	2F840
Offset:	0000 0101 0011 0010	=	00532
<hr/>			
address:	0010 1111 1101 0111 0010	=	2FD72



A chunk of 16 bytes is known as a *paragraph*, so you could say that a segment always starts on a paragraph boundary.

The starting address of a segment is always a 20-bit number, but a segment register only holds 16 bits—so the bottom 4 bits are always assumed to be all zeros. This means segments can only start every 16 bytes through memory, at an address where the last 4 bits (or last hex digit) are zero. So, if the DS register is holding a value of 2F84, then the data segment actually starts at address 2F840.

The standard notation for an address takes the form *segment:offset*; for example, the previous address would be written as 2F84:0532. Note that since offsets can overlap, a given segment:offset pair is not unique; the following addresses all refer to the same memory location:

```
0000:0123
0002:0103
0008:00A3
0010:0023
0012:0003
```

Segments can overlap (but don't have to). For example, all four segments could start at the same address, which means that your entire program would take up no more than 64K—but that's all the space you'd have for your code, your data, and your stack.

Pointers

Although you can declare a pointer or function to be a specific type regardless of the model used, by default the type of memory model you choose determines the default type of pointers used for code and data. There are four types of pointers: *near* (16 bits), *far* (32 bits), *huge* (also 32 bits), and *segment* (16 bits).

Near pointers

A near pointer (16-bits) relies on one of the segment registers to finish calculating its address; for example, a pointer to a function would add its 16-bit value to the left-shifted contents of the code segment (CS) register. In a similar fashion, a near data pointer contains an offset to the data segment (DS) register. Near pointers are easy to manipulate, since any arithmetic (such as addition) can be done without worrying about the segment.

Far pointers

A far pointer (32-bits) contains not only the offset within the segment, but also the segment address (as another 16-bit value), which is then left-shifted and added to the offset. By using far pointers, you can have multiple code segments; this, in turn, allows you to have programs larger than 64K. You can also address more than 64K of data.

When you use far pointers for data, you need to be aware of some potential problems in pointer manipulation. As explained in the section on address calculation, you can have many different segment:offset pairs refer to the same address. For example, the far pointers 0000:0120, 0010:0020, and 0012:0000 all resolve to the same 20-bit address. However, if you had three different far pointer variables—*a*, *b*, and *c*—containing those three values respectively, then all the following expressions would be *false*:

```
if (a == b) . . .  
if (b == c) . . .  
if (a == c) . . .
```

A related problem occurs when you want to compare far pointers using the *>*, *>=*, *<*, and *<=* operators. In those cases, only the offset (as an **unsigned**) is used for comparison purposes; given that *a*, *b*, and *c* still have the values previously listed, the following expressions would all be *true*:

```
if (a > b) . . .  
if (b > c) . . .  
if (a > c) . . .
```

The equals (*=*) and not-equal (*!=*) operators use the 32-bit value as an **unsigned long** (not as the full memory address). The comparison operators (*<=*, *>=*, *<*, and *>*) use just the offset.

The *=* and *!=* operators need all 32 bits, so the computer can compare to the NULL pointer (0000:0000). If you used only the offset value for equality checking, any pointer with 0000 offset would be equal to the NULL pointer, which is not what you want.



If you add values to a far pointer, only the offset is changed. If you add enough to cause the offset to exceed FFFF (its maximum possible value), the pointer just wraps around back to the beginning of the segment. For example, if you add 1 to 5031:FFFF, the result would be 5031:0000 (not 6031:0000). Likewise, if you subtract 1 from 5031:0000, you would get 5031:FFFF (not 5030:000F).

If you want to do pointer comparisons, it's safest to use either near pointers—which all use the same segment address—or huge pointers, described next.

Huge pointers

Huge pointers are also 32 bits long. Like far pointers, they contain both a segment address and an offset. Unlike far pointers, they are *normalized* to avoid the problems associated with far pointers.

A normalized pointer is a 32-bit pointer that has as much of its value in the segment address as possible. Since a segment can start every 16 bytes (10 in base 16), this means that the offset will only have a value from 0 to 15 (0 to F in base 16).

To normalize a pointer, convert it to its 20-bit address, then use the right 4 bits for your offset and the left 16 bits for your segment address. For example, given the pointer 2F84:0532, you would convert that to the absolute address 2FD72, which you would then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

0000:0123	0012:0003
0040:0056	0045:0006
500D:9407	594D:0007
7418:D03F	811B:000F

There are three reasons why it is important to always keep huge pointers normalized:

1. For any given memory address there is only one possible huge address (segment:offset) pair. That means that the `==` and `!=` operators return correct answers for any huge pointers.
2. In addition, the `>`, `>=`, `<`, and `<=` operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results of these comparisons will also be correct.
3. Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment 811B:000F, the result would be 811C:0000; likewise, if you decrement 811C:0000, you get 811B:000F. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size. This ensures that, for example, if you have a huge array of **structs** that is larger than 64K, indexing into the array and selecting a **struct** field will always work with structs of any size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

The five memory models

Paradigm C++ gives you five memory models for 16-bit DOS programs: small, medium, compact, large, and huge. Your program requirements determine which one you pick. Here's a brief summary of each:

- **Small.** The code and data segments are different and don't overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used. This is a good size for average applications.
- **Medium.** Far pointers are used for code, but not for data. As a result, data plus stack are limited to 64K, but code can occupy up to 1 MB. This model is best for large programs without much data in memory.
- **Compact.** The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1 MB range. This model is best if code is small but needs to address a lot of data.
- **Large.** Far pointers are used for both code and data, giving both a 1 MB range. Large and huge are needed only for very large applications.

- **Huge.** Far pointers are used for both code and data. Paradigm C++ normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing data to occupy more than 64K.

The following figures show how memory in the 8086 is apportioned for the Paradigm C++ memory models. To select these memory models, you can either use menu selections from the IDE or you can type options invoking the Paradigm C++ command-line compiler.

Figure 13-3
Small model
memory
segmentation

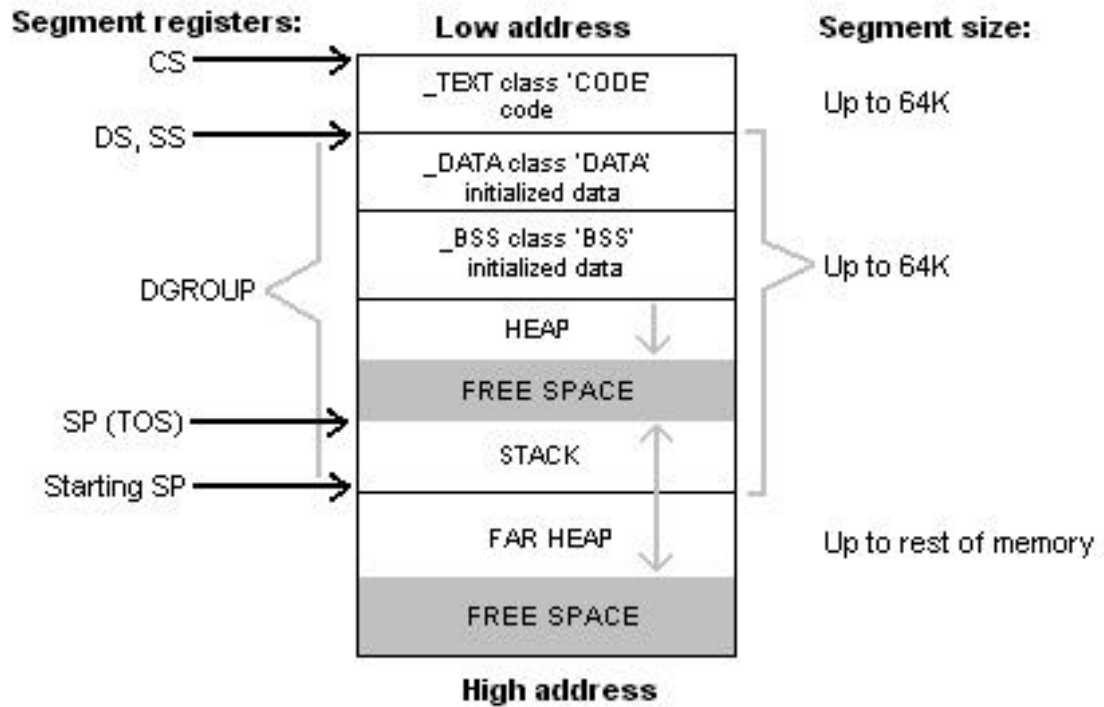


Figure 13-4
Medium model
memory
segmentation

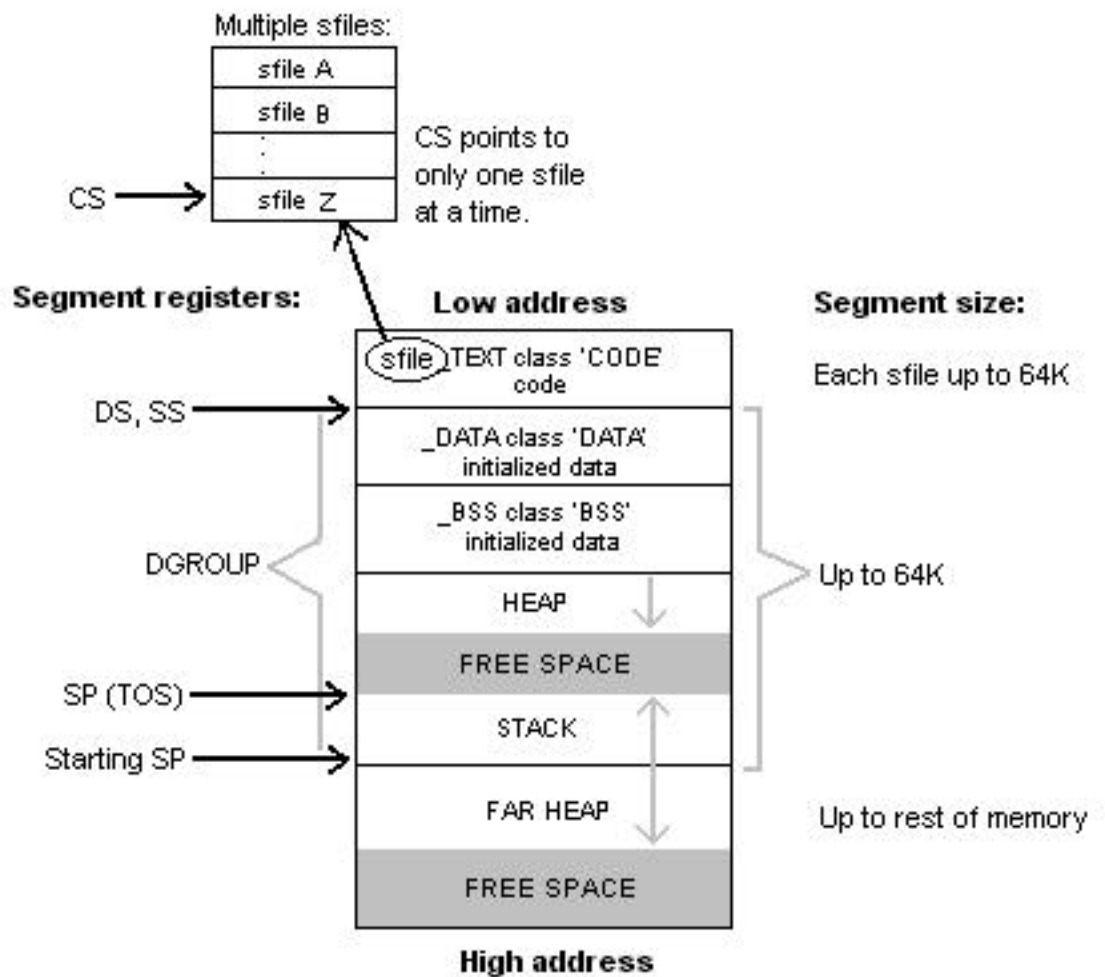


Figure 13-5
Compact model
memory
segmentation

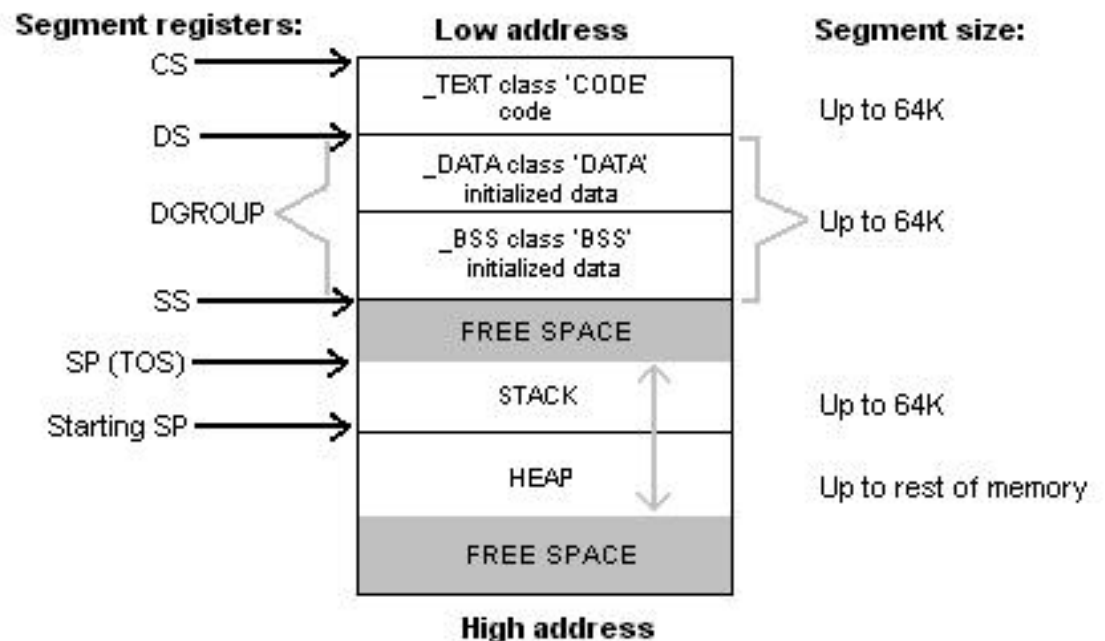


Figure 13-6
Large model
memory
segmentation

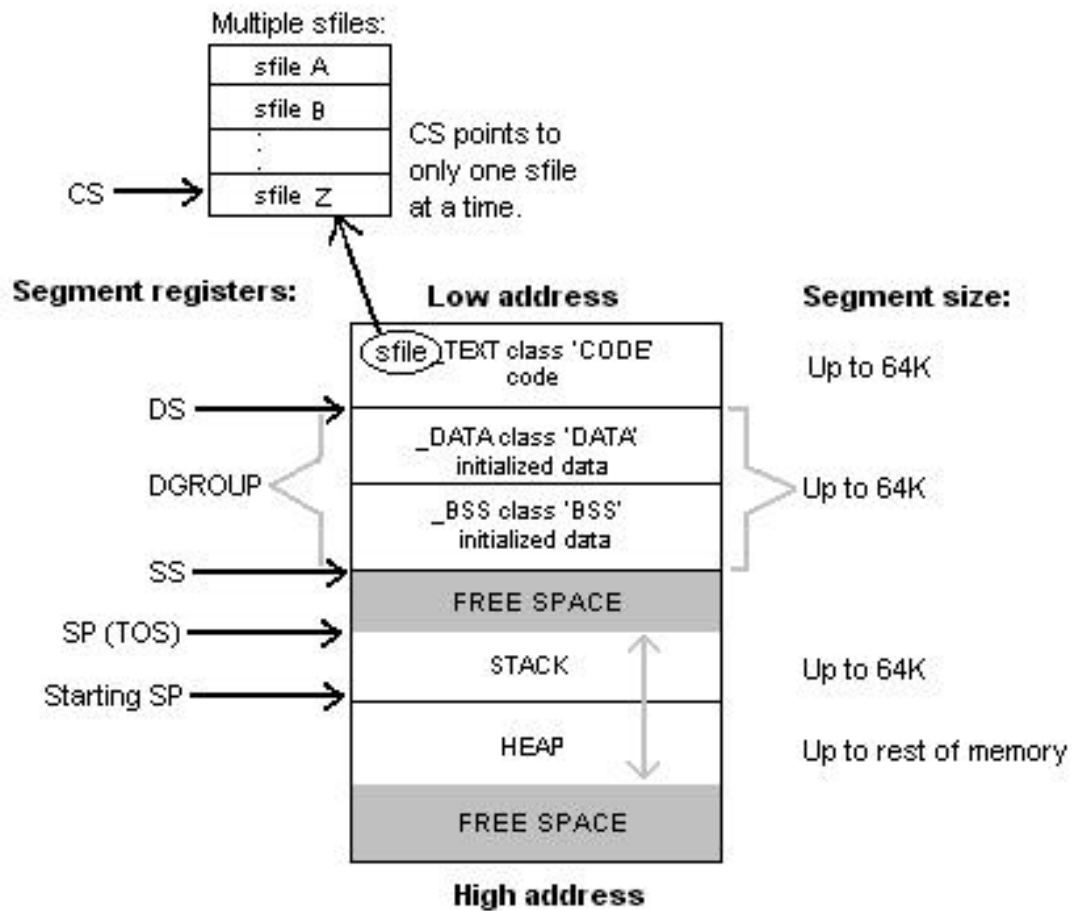
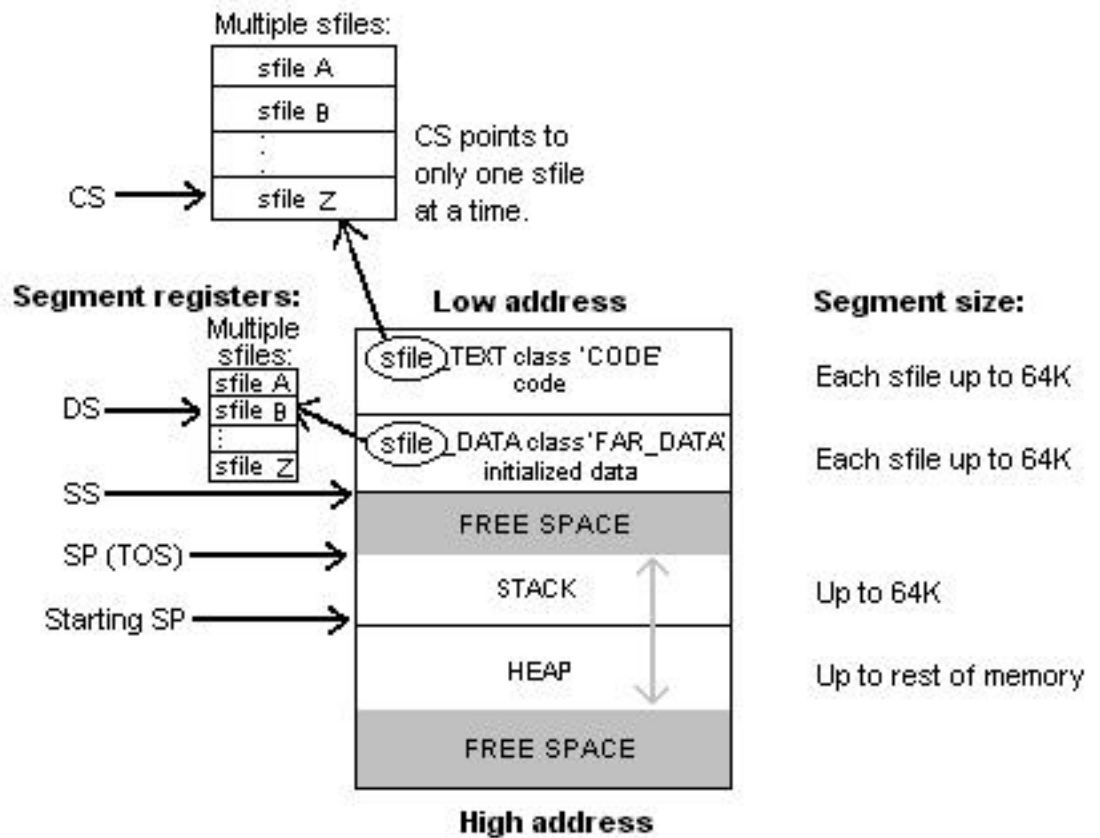


Figure 13-7
Huge model
memory
segmentation



The following table summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are *small* (64K) or *large* (16 MB); these groups correspond to the rows and columns in the table.

Table 13-1
Comparison of
models

Data size	Code size = 64K	Code size = 16MB
64K	Small (no overlap; total size = 128K)	Medium (small data, large code)
16 MB	Compact (large data, small code)	Large (large data, code)
		Huge (same as large but static data > 64K)

The small and compact models are small code models because, by default, code pointers are near; likewise, compact, large, and huge are large data models because, by default, data pointers are far.

When you compile a module (a given source file with some number of routines in it), the resulting code for that module cannot be greater than 64K, since it must all fit inside of one code segment. This is true even if you're using one of the larger code models (medium, large, or huge). If your module is too big to fit into one (64K) code segment, you must break it up into different source code files, compile each file separately, then link them together. Similarly, even though the huge model permits static data to total more than 64K, it still must be less than 64K in each module.

Mixed-model programming: Addressing modifiers

Paradigm C++ introduces eight new keywords not found in standard ANSI C. These keywords are `__near`, `__far`, `__huge`, `__cs`, `__ds`, `__es`, `__ss`, and `__seg`. These keywords can be used as modifiers to pointers (and in some cases, to functions), with certain limitations and warnings.

In Paradigm C++, you can modify the declarations of pointers, objects, and functions with the keywords `__near`, `__far`, or `__huge`. The `__near`, `__far`, and `__huge` data pointers are described in “Pointers,” page 13-245. You can declare far objects using the `__far` keyword. `__near` functions are invoked with near calls and exit with near returns. Similarly, `__far` functions are called `__far` and return far values. `__huge` functions are like `__far` functions, except that `__huge` functions set DS to a new value, and `__far` functions do not.

There are also four special `__near` data pointers: `__cs`, `__ds`, `__es`, and `__ss`. These are 16-bit pointers that are specifically associated with the corresponding segment register. For example, if you were to declare a pointer to be

```
char __ss *p;
```

Then `p` would contain a 16-bit offset into the stack segment.

Functions and pointers within a given program default to near or far, depending on the memory model you select. If the function or pointer is near, it is automatically associated with either the CS or DS register.

The following table shows how this works. Note that the size of the pointer corresponds to whether it is working within a 64K memory limit (near, within a segment) or inside the general 1 MB memory space (far, has its own segment address).

Table 13-2
Defaults for
functions and
pointers

Memory model	Function pointers	Data pointers
Small	near, <code>__cs</code>	near, <code>__ds</code>
Medium	far	near, <code>__ds</code>
Compact	near, <code>__cs</code>	far
Large	far	far
Huge	far	far

Segment pointers

Use `__seg` in segment pointer type declarators. The resulting pointers are 16-bit segment pointers. The syntax for `__seg` is:

```
datatype __seg *identifier;
```

For example,

```
int __seg *name;
```

Any indirection through *identifier* has an assumed offset of 0. In arithmetic involving segment pointers the following rules hold true:

1. You can't use the `++`, `--`, `+=`, or `-=` operators with segment pointers.
2. You cannot subtract one segment pointer from another.
3. When adding a near pointer to a segment pointer, the result is a far pointer that is formed by using the segment from the segment pointer and the offset from the near pointer. Therefore, the two pointers must either point to the same type, or one must

be a pointer to void. There is no multiplication of the offset regardless of the type pointed to.

4. When a segment pointer is used in an indirection expression, it is also implicitly converted to a far pointer.
5. When adding or subtracting an integer operand to or from a segment pointer, the result is a far pointer, with the segment taken from the segment pointer and the offset found by multiplying the size of the object pointed to by the integer operand. The arithmetic is performed as if the integer were added to or subtracted from the far pointer.
6. Segment pointers can be assigned, initialized, passed into and out of functions, compared and so forth. (Segment pointers are compared as if their values were **unsigned** integers). In other words, other than the above restrictions, they are treated exactly like any other pointer.

Declaring far objects

You can declare far objects in Paradigm C++. For example,

```
int far x = 5;
int far z;
extern int far y = 4;
static long j;
```

The command-line compiler options **-zE**, **-zF**, and **-zH** (which can also be set using **#pragma option**) affect the far segment name, class, and group, respectively. When you use **#pragma option**, you can make them apply to any ensuing far object declarations. Thus you could use the following sequence to create a far object in a specific segment:

```
#pragma option -zEmysegment -zHmygroup -zFmyclass
int far x;
#pragma option -zE* -zH* -zF*
```

This will put *x* in segment MYSEGMENT 'MYCLASS' in the group 'MYGROUP', then reset all of the far object items to the default values. Note that by using these options, several far objects can be forced into a single segment:

```
#pragma option -zEcombined -zFmyclass
int far x;
double far y;
#pragma option -zE* -zF*
```

Both *x* and *y* will appear in the segment COMBINED 'MYCLASS' with no group.

Declaring functions to be near or far

On occasion, you'll want (or need) to override the default function type of your memory model.

For example, suppose you're using the large memory model, but you have a recursive (self-calling) function in your program, like this:

```
double power(double x,int exp) {
    if (exp <= 0)
        return(1);
    else
        return(x * power(x, exp-1));
}
```

Every time *power* calls itself, it has to do a far call, which uses more stack space and clock cycles. By declaring *power* as `__near`, you eliminate some of the overhead by forcing all calls to that function to be near:

```
double __near power(double x,int exp)
```

This guarantees that *power* is callable only within the code segment in which it was compiled, and that all calls to it are near calls.

This means that if you're using a large code model (medium, large, or huge), you can only call *power* from within the module where it is defined. Other modules have their own code segment and thus cannot call `__near` functions in different modules. Furthermore, a near function must be either defined or declared before the first time it is used, or the compiler won't know it needs to generate a near call.

Conversely, declaring a function to be far means that a far return is generated. In the small code models, the far function must be declared or defined before its first use to ensure it is invoked with a far call.

Look back at the *power* example at the beginning of this section. It is wise to also declare *power* as static, since it should be called only from within the current module. That way, being a static, its name will not be available to any functions outside the module.

Declaring pointers to be near, far, or huge

You've seen why you might want to declare functions to be of a different model than the rest of the program. For the same reasons given in the preceding section, you might want to modify pointer declarations: either to avoid unnecessary overhead (declaring `__near` when the default would be `__far`) or to reference something outside of the default segment (declaring `__far` or `__huge` when the default would be `__near`).

There are, of course, potential pitfalls in declaring functions and pointers to be of non-default types. For example, say you have the following small model program:

```
void myputs(s) {
    char *s;
    int i;
    for (i = 0; s[i] != 0; i++) putc(s[i]);
}

main() {
    char near *mystr;

    mystr = "Hello, world\n";
    myputs(mystr);
}
```

This program works fine. In fact, the `__near` declaration on *mystr* is redundant, since all pointers, both code and data, will be near.

But what if you recompile this program using the compact (or large or huge) memory model? The pointer *mystr* in *main* is still near (it's still a 16-bit pointer). However, the pointer *s* in *myputs* is now far, because that's the default. This means that *myputs* will pull two words out of the stack in an effort to create a far pointer, and the address it ends up with will certainly not be that of *mystr*.

How do you avoid this problem? If you're going to explicitly declare pointers to be of type `__far` or `__near`, be sure to use function prototypes for any functions that might use them. The solution is to define *myputs* in ANSI C style, like this:

```
void myputs(char *s) {
    /* body of myputs */
}
```

Now when Paradigm C++ compiles your program, it knows that *myputs* expects a pointer to **char**; and since you are compiling under the large model, it knows that the pointer must be **__far**. Because of that, Paradigm C++ will push the data segment (DS) register onto the stack along with the 16-bit value of *mystr*, forming a far pointer.

How about the reverse case: arguments to *myputs* declared as **__far** and compiled with a small data model? Again, without the function prototype, you will have problems, because *main* will push both the offset and the segment address onto the stack, but *myputs* will expect only the offset. With the prototype-style function definitions, though, *main* will only push the offset onto the stack.

Pointing to a given segment:offset address

You can make a far pointer point to a given memory location (a specific segment:offset address). You can do this with the macro *MK_FP*, which takes a segment and an offset and returns a far pointer. For example,

```
MK_FP(segment_value, offset_value)
```



Given a **__far** pointer, *fp*, you can get the segment component with *FP_SEG(fp)* and the offset component with *FP_OFF(fp)*.

Using library files



Paradigm C++ offers a version of the standard library routines for each of the five memory models. Paradigm C++ is smart enough to link in the appropriate libraries in the proper order, depending on which model you've selected. However, if you're using the Paradigm C++ linker, PLINK, directly (as a stand-alone linker), you need to specify which libraries to use. See "Using PLINK" in the online Help index for instructions on how to do this.

Linking mixed modules

Suppose you compiled one module using the small memory model and another module using the large model, then wanted to link them together. This would present some problems, but they can be solved.

The files would link together fine, but the problems you would encounter would be similar to those described in the section, "Declaring functions to be near or far," page 13-253. If a function in the small module called a function in the large module, it would do so with a near call, which would probably be disastrous. Furthermore, you could face the same problems with pointers as described in "Declaring pointers to be near, far, or huge," page 13-254, since a function in the small module would expect to pass and receive **__near** pointers, and a function in the large module would expect **__far** pointers.

The solution, again, is to use function prototypes. Suppose that you put *myputs* into its own module and compile it with the large memory model. Then create a header file called *myputs.h* (or some other name with a .h extension), which would have the following function prototype in it:

```
void far myputs(char far *s);
```

Now, put *main* into its own module (called *MYMAIN.C*), and set things up like this:

```

#include <stdio.h>
#include "myputs.h"

main() {
    char near *mystr;

    mystr = "Hello, world\n";
    myputs(mystr);
}

```

When you compile this program, Paradigm C++ reads in the function prototype from `myputs.h` and sees that it is a `__far` function that expects a `__far` pointer. Therefore, it generates the proper calling code, even if it's compiled using the small memory model.

What if, on top of all this, you need to link in library routines? Your best bet is to use one of the large model libraries and declare everything to be `__far`. To do this, make a copy of each header file you would normally include (such as `stdio.h`), and rename the copy to something appropriate (such as `fstdio.h`).

Then edit each function prototype in the copy so that it is explicitly `__far`, like this:

```
int far cdecl printf(char far * format, ...);
```

That way, not only will `__far` calls be made to the routines, but the pointers passed will also be `__far` pointers. Modify your program so that it includes the new header file:

```

#include <fstdio.h>

void main() {
    char near *mystr;
    mystr = "Hello, world\n";
    printf(mystr);
}

```

Compile your program with the command-line compiler PCC then link it with PLINK, specifying a large model library, such as `CL.LIB`. Mixing models is tricky, but it can be done; just be prepared for some difficult bugs if you do things wrong.

Using iostreams classes

Paradigm provides a full implementation of the C++ input and output classes, commonly known as iostreams. With the arrival of C++ and object-oriented design, input and output operations became encapsulated in a series of classes. Each iostreams class encapsulates some form of input, output, or input and output from low-level character transfer to higher-level, file-oriented input/output operations.

Stream input/output in C++ (commonly referred to as *iostreams*, or just *streams*) provides all the functionality of the *stdio* library in ANSI C and much more. Iostreams are used to convert typed objects into readable text, and vice versa. Streams can also read and write binary data. The C++ language lets you define or overload I/O functions and operators that are then called automatically for corresponding user-defined types.

What is a stream?

A stream is an abstraction referring to any flow of data from a source (or *producer*) to a *sink* (or *consumer*). We also use the synonyms *extracting*, *getting*, and *fetching* when speaking of inputting characters from a source; and *inserting*, *putting*, or *storing* when speaking of outputting characters to a sink. Classes are provided that support console output (*constrea.h*), memory buffers (*iostream.h*), files (*fstream.h*), and strings (*strstrea.h*) as sources or sinks (or both).

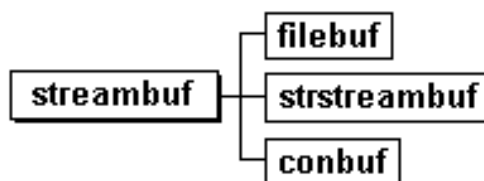
The iostream library

The *iostream* library has two parallel families of classes: those derived from *streambuf*, and those derived from *ios*. Both are low-level classes, each doing a different set of jobs. All stream classes have at least one of these two classes as a base class. Access from *ios*-based classes to *streambuf*-based classes is through a pointer.

The streambuf class

The *streambuf* class provides an interface to memory and physical devices. *streambuf* provides underlying methods for buffering and handling streams when little or no formatting is required. The member functions of the *streambuf* family of classes are used by the *ios*-based classes. You can also derive classes from *streambuf* for your own functions and libraries. The buffering classes *conbuf*, *filebuf*, and *strstreambuf* are derived from *streambuf*.

Figure 14-1
Class *streambuf*
and its derived
classes



The ios class

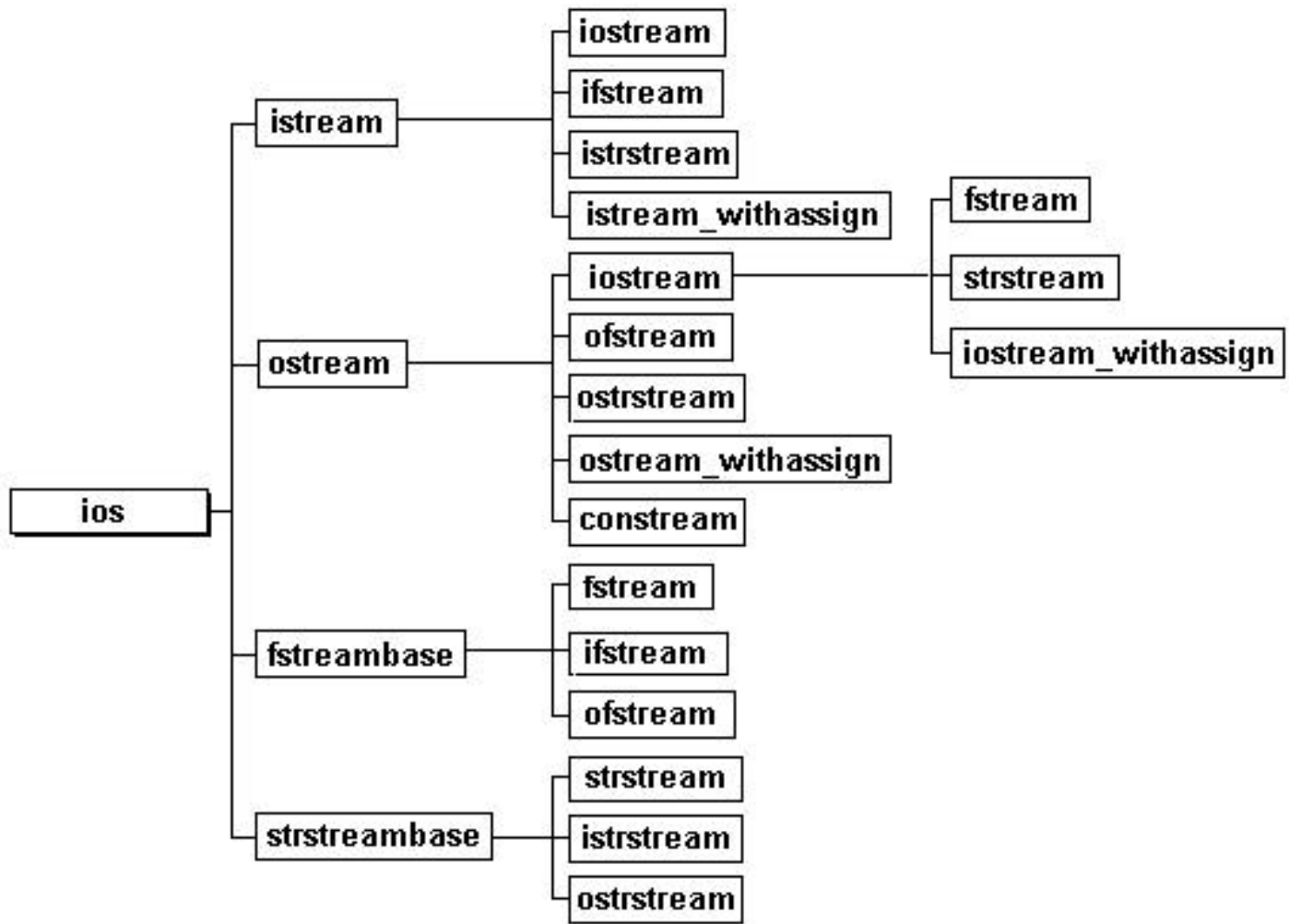
The class *ios* (and hence any of its derived classes) contains a pointer to a *streambuf*. It performs formatted I/O with error-checking using a *streambuf*.

An inheritance diagram for all the *ios* family of classes is found in Figure 14-2, page 14-259. For example, the *ifstream* class is derived from the *istream* and *fstreambase* classes, and *istrstream* is derived from *istream* and *strstreambase*. This diagram is not a simple hierarchy because of the generous use of *multiple inheritance*. With multiple inheritance, a single class can inherit from more than one base class. (The C++ language provides for *virtual inheritance* to avoid multiple declarations.) This means, for example, that all the members (data and functions) of *iostream*, *istream*, *ostream*, *fstreambase*, and *ios* are part of objects of the *fstream* class. All classes in the *ios*-based tree use a *streambuf* (or a *filebuf* or *strstreambuf*, which are special cases of a *streambuf*) as its source and/or sink.

C++ programs start with four predefined open streams, declared as objects of *withassign* classes as follows:

```
extern istream_withassign cin; // Corresponds to stdin;  
                               file descriptor 0.  
extern ostream_withassign cout; // Corresponds to stdout;  
                                file descriptor 1.  
extern ostream_withassign cerr; // Corresponds to stderr;  
                                file descriptor 2.  
extern ostream_withassign clog; // A buffered cerr;  
                                file descriptor 2.
```

Figure 14-2 Class ios and its derived classes



By accepted practice, the arrows point **from** the derived class **to** the base class.

Stream output

Stream output is accomplished with the *insertion* (or *put to*) operator, `<<`. The standard left shift operator, `<<`, is overloaded for output operations. Its left operand is an object of type *ostream*. Its right operand is any type for which stream output has been defined (that is, fundamental types or any types you have overloaded it for). For example,

```
cout << "Hello!\n";
```

writes the string "Hello!" to *cout* (the standard output stream, normally your screen) followed by a new line.

The `<<` operator associates from left to right and returns a reference to the *ostream* object it is invoked for. This allows several insertions to be cascaded as follows:

```
int i = 8;
double d = 2.34;
cout << "i = " << i << ", d = " << d << "\n";
```

This will write the following to standard output:

```
i = 8, d = 2.34
```

Fundamental types

The fundamental data types directly supported are **char**, **short**, **int**, **long**, **char*** (treated as a string), **float**, **double**, **long double**, and **void***. Integral types are formatted according to the default rules for *printf* (unless you've changed these rules by setting various *ios* flags). For example, the following two output statements give the same result:

```
int i;
long l;
cout << i << " " << l;
printf("%d %ld", i, l);
```

The pointer (**void ***) inserter is used to display pointer addresses:

```
int i;
cout << &i;           // display pointer address in hex
```



For more information, read the description of "*ostream*" in the online Help Book Shelf index. The Book Shelf index can be accessed by choosing Help|Keyboard and clicking the Book Shelf menu tab.

I/O formatting

Formatting for both input and output is determined by various *format state* flags contained in the **class ios**. The flags are read and set with the *flags*, *setf*, and *unsetf* member functions.

Output formatting can also be affected by the use of the *fill*, *width*, and *precision* member functions of **class ios**.



The format flags are detailed in the description of "ios class" in the online Help Book Shelf index. The Book Shelf index can be accessed by choosing Help|Keyboard and clicking the Book Shelf menu tab.

Manipulators

A simple way to change some of the format variables is to use a special function-like operator called a *manipulator*. Manipulators take a stream reference as an argument and return a reference to the same stream. You can embed manipulators in a chain of insertions (or extractions) to alter stream states as a side effect without actually performing any insertions (or extractions). Parameterized manipulators must be called for each stream operation. For example,

```
#include <iostream.h>
#include <iomanip.h>  // Required for parameterized manipulators.

int main(void) {
    int i = 6789, j = 1234, k = 10;

    cout << setw(6) << i << j << i << k << j;
    cout << "\n";
    cout << setw(6) << i << setw(6) << j << setw(6) << k;
    return(0);
}
```

produces this output:

```
678912346789101234
6789 1234 10
```

setw is a parameterized manipulator declared in *iomanip.h*. Other parameterized manipulators, *setbase*, *setfill*, *setprecision*, *setiosflags* and *resetiosflags*, work in the same way. To make use of these, your program must include *iomanip.h*. You can write your own manipulators without parameters:

```
#include <iostream.h>

// Tab and prefix the output with a dollar sign.
ostream& money( ostream& output) {
    return output << "\t$";
}

int main(void) {
    float owed = 1.35, earned = 23.1;
    cout << money << owed << money << earned;
    return(0);
}
```

produces the following output:

```
$1.35    $23.1
```

The non-parameterized manipulators *dec*, *hex*, and *oct* (declared in *iostream.h*) take no arguments and simply change the conversion base (and leave it changed):

```
int i = 36;
cout << dec << i << " " << hex << i << " " << oct << i << endl;
cout << dec; // Must reset to use decimal base.
// displays 36 24 44
```

Table 14-1
Stream
manipulators

Manipulator	Action
<i>dec</i>	Set decimal conversion base format flag.
<i>hex</i>	Set hexadecimal conversion base format flag.
<i>oct</i>	Set octal conversion base format flag.
<i>ws</i>	Extract whitespace characters.
<i>endl</i>	Insert newline and flush stream.
<i>ends</i>	Insert terminal null in string.
<i>flush</i>	Flush an ostream.
<i>setbase(int n)</i>	Set conversion base format to base n (0, 8, 10, or 16). 0 means the default: decimal on output, ANSI C rules for literal integers on input.
<i>resetiosflags(long f)</i>	Clear the format bits specified by <i>f</i> .
<i>setiosflags(long f)</i>	Set the format bits specified by <i>f</i> .
<i>setfill(int c)</i>	Set the fill character to <i>c</i> .
<i>setprecision(int n)</i>	Set the floating-point precision to <i>n</i> .
<i>setw(int n)</i>	Set field width to <i>n</i> .

The manipulator *endl* inserts a newline character and flushes the stream. You can also flush an *ostream* at any time with

```
ostream << flush;
```

Filling and padding

The fill character and the direction of the padding depend on the setting of the fill character and the left, right, and internal flags.

The default fill character is a space. You can vary this by using the function *fill*:

```
int i = 123;
cout.fill('*');
cout.width(6);
cout << i;           // display ***123
```

The default direction of padding gives right-alignment (pad on the left). You can vary these defaults (and other format flags) with the functions *setf* and *unsetf*:

```
int i = 56;
.
.
.
cout.width(6);
cout.fill('#');
cout.setf(ios::left, ios::adjustfield);
cout << i;           // display 56####
```



The second argument, *ios::adjustfield*, tells *setf* which bits to set. The first argument, *ios::left*, tells *setf* what to set those bits to. Alternatively, you can use the manipulators *setfill*, *setiosflags*, and *resetiosflags* to modify the fill character and padding mode. See "ios:adjustfield" in the online Help Book Shelf index, for a list of masks used by *setf*. The Book Shelf index can be accessed by choosing Help|Keyboard and clicking the Book Shelf menu tab.

Stream input

Stream input is similar to output but uses the overloaded right shift operator, *>>*, known as the *extraction* (get from) operator or *extractor*. The left operand of *>>* is an object of type **class** *istream*. As with output, the right operand can be of any type for which stream input has been defined.

By default, *>>* skips whitespace (as defined by the *isspace* function in *ctype.h*), then reads in characters appropriate to the type of the input object. Whitespace skipping is controlled by the *ios::skipws* flag in the format state's enumeration. The *skipws* flag is normally set to give whitespace skipping. Clearing this flag (with *setf*, for example) turns off whitespace skipping. There is also a special "sink" manipulator, *ws*, that lets you discard whitespace.

Consider the following example:

```
int i;
double d;
cin >> i >> d;
```

When the last line is executed, the program skips any leading whitespace. The integer value (*i*) is then read. Any whitespace following the integer is ignored. Finally, the floating-point value (*d*) is read.



For type **char** (**signed** or **unsigned**), the effect of the *>>* operator is to skip whitespace and store the next (non-whitespace) character. If you need to read the next character, whether it is whitespace or not, you can use one of the *get* member functions. See the discussion of "istream" in online Help Book Shelf index. The Book Shelf index can be accessed by choosing Help|Keyboard and clicking the Book Shelf menu tab.

For type **char*** (treated as a string), the effect of the *>>* operator is to skip whitespace and store the next (non-whitespace) characters until another whitespace character is found. A final null character is then appended. Care is needed to avoid "overflowing" a string. You can alter the default width of zero (meaning no limit) using *width* as follows:

```

char array[SIZE];
cin.width(sizeof(array));
cin >> array;           // Avoids overflow.

```

For all input of fundamental types, if only whitespace is encountered, nothing is stored in the target, and the istream state is set to *fail*. The target will retain its previous value; if it was uninitialized, it remains uninitialized.

I/O of user-defined types

To input or output your own defined types, you must overload the extraction and insertion operators. Here is an example:

```

#include <iostream.h>

struct info {
    char *name;
    double val;
    char *units;
};

// You can overload << for output as follows:
ostream& operator << (ostream& s, info& m) {
    s << m.name << " " << m.val << " " << m.units;
    return s;
};

// You can overload >> for input as follows:
istream& operator >> (istream& s, info& m) {
    s >> m.name >> m.val >> m.units;
    return s;
};

int main(void) {
    info x;
    x.name = new char[15];
    x.units = new char[10];

    cout << "\nInput name, value and units:";
    cin >> x;
    cout << "\nMy input:" << x;
    return(0);
}

```

Simple file I/O

The class *ofstream* inherits the insertion operations from *ostream*, while *ifstream* inherits the extraction operations from *istream*. The file-stream classes also provide constructors and member functions for creating files and handling file I/O. You must include *fstream.h* in all programs using these classes.

Consider the following example that copies the file FILE.IN to the file FILE.OUT:

```

#include <fstream.h>

int main(void) {
    char ch;
    ifstream f1("FILE.IN");
    ofstream f2("FILE.OUT");
}

```

```

    if (!f1) cerr << "Cannot open FILE.IN for input";
    if (!f2) cerr << "Cannot open FILE.OUT for output";
    while (f2 && f1.get(ch))
        f2.put(ch);
    return(0);
}

```

Note that if the *ifstream* or *ofstream* constructors are unable to open the specified files, the appropriate stream error state is set.

The constructors let you declare a file stream without specifying a named file. Later, you can associate the file stream with a particular file:

```

ofstream ofile;           // creates output file stream
.
.
.
ofile.open("payroll");    // ofile connects to file "payroll"
// do some payrolling...
ofile.close();            // close the ofile stream
ofile.open("employee");   // ofile can be reused...

```



By default, files are opened in text mode. This means that on input, carriage-return/linefeed sequences are converted to the '\n' character. On output, the '\n' character is converted to a carriage-return/linefeed sequence. These translations are not done in binary mode. The file-opening mode is set with an optional second parameter to the *open* function or in some file-stream constructors. The file opening-mode constraints can be used alone or they can logically ORed together. See the description of "ios class" in the online Help Book Shelf index. The Book Shelf index can be accessed by choosing Help|Keyboard and clicking the Book Shelf menu tab.

String stream processing

The functions defined in *strstrea.h* support in-memory formatting, similar to *sscanf* and *sprintf*, but much more flexible. All of the *istream* member functions are available for **class** *istrstream* (input string stream). This is the same for output: *ostrstream* inherits from *ostream*.

Given a text file with the following format:

```

101 191 Cedar Chest
102 1999.99 Livingroom Set

```

Each line can be parsed into three components: an integer ID, a floating-point price, and a description. The output produced is

```

1: 101 191.00 Cedar Chest
2: 102 1999.99 Livingroom Set

```

Here is the program:

```

#include <fstream.h>
#include <strstrea.h>
#include <iomanip.h>
#include <string.h>

```



```

int main(int argc, char **argv) {
    int id;
    float amount;
    char description[41];

    if (argc == 1) {
        cout << "\nInput file name required.";
        return (-1);
    }

    ifstream inf(argv[1]);

    if (inf) {
        char inbuf[81];
        int lineno = 0;

        // Want floats to print as fixed point
        cout.setf(ios::fixed, ios::floatfield);

        // Want floats to always have decimal point
        cout.setf(ios::showpoint);

        while (inf.getline(inbuf,81)) {
            // 'ins' is the string stream:
            istrstream ins(inbuf,strlen(inbuf));
            ins >> id >> amount >> ws;
            ins.getline(description,41); // Linefeed not copied.
            cout << ++lineno << ": "
                << id << '\t'
                << setprecision(2) << amount << '\t'
                << description << "\n";
        }
    }
    return(0);
}

```

Note the use of format flags and manipulators in this example. The calls to *setf* coupled with *setprecision* allow floating-point numbers to be printed in a money format. The manipulator *ws* skips whitespace before the description string is read.

Errors and messages

This appendix describes the error messages that can be generated by Paradigm C++. It begins by describing the four types of messages you can receive: fatal errors, errors, warnings, and informational messages.

Next, it covers the different components that can generate messages: the compiler, the MAKE utility, the linker (PLINK), the librarian (PLIB), the integrated debugger, and the Windows Help compiler. This appendix also lists the errors that you can receive when you run your program (run-time errors).

The remainder of the appendix lists messages in ASCII alphabetic order and provides a description of each message that includes where the message was generated.

Message categories

Messages are displayed with the message class first, followed by the source file name and line number where the error was detected, and finally with the text of the message itself.

The following categories of messages can occur:

Table A-1
Message
categories

Category	Indicates
Fatal	A problem of critical nature that prevents execution from continuing.
Error	A problem that should be fixed such as a missing declaration or a type mismatch.
Warning	A problem that can be overlooked.
Informational	Progress such as build status.

Many of the messages appear in the Message view. For those messages, context-sensitive help is available. Point to the message and press *F1* to display the message description.



If you are working from the command line or want to look up information on an error message, refer to the alphabetical list of error and warning messages in "Alphabetical list of messages" later on in this chapter. A listing is also available in the online Help Book Shelf under "Paradigm C++ error and warnings". The Book Shelf can be accessed by choosing Help|Keyboard and clicking the Book Shelf menu tab.

Fatal errors

Fatal errors can be generated by the compiler, the linker, and the MAKE utility. Fatal errors cause the compilation to stop immediately; you must take appropriate action to fix the error before you resume compiling.

If the compiler or MAKE utility issues a fatal error, no .AXE files is created. If the linker issues a fatal error, any .AXE file that might have been created by the linker is deleted before the linker returns.

Errors

Errors can be generated by the compiler, the linker, and the MAKE utility, and the librarian. In addition, errors can be generated by your program at run-time.

Errors generated by the compiler indicate program syntax errors, command-line errors, and disk or memory access errors. Compiler errors don't cause the compilation to stop - the compiler completes the current phase of the compilation and then stops and reports the errors encountered. The compiler attempts to find as many real errors in the source program as possible during each phase (preprocessing, parsing, optimizing, and code-generating).

Errors generated by the linker don't cause the linker to delete the .AXE or .MAP files. However, you shouldn't execute any .AXE file that was linked with errors. Linker errors are treated like fatal errors if you are compiling from the Paradigm C++ IDE.

The MAKE utility generates errors when there is a syntax or semantic error in the source makefile. You must edit the makefile to fix these types of errors.

Run-time errors are usually caused by logic errors in your program code. If you receive a run-time error, you must fix the error in your source code and recompile the program for the fix to take effect.

Warnings

Warnings can be issued by the compiler, the linker, and the librarian. Warnings do not prevent the compilation from finishing. However, they do indicate conditions that are suspicious, even if the condition that caused the warnings is legitimate within the language. The compiler also produces warnings if you use machine-dependent constructs in your source files.

Informational messages

Informational messages inform you about the progress of tasks such as the status of a build.

Message generators

The messages in this appendix include messages that can be generated by the compiler, the MAKE utility, the linker (PLINK), the librarian, (PLIB), the Paradigm C++ IDE, and the Windows Help compiler. Run-time errors (errors you can receive when you run your program) are also included.

Compiler errors and warnings

Compile-time error messages indicate errors in program syntax, command-line errors, or errors in accessing a disk or memory. When most compile-time errors occur, the compiler completes the current phase (preprocessing, parsing, optimizing, and code-generating) of the compilation and stops. But when fatal compile-time errors happen, compilation stops completely. If a fatal error occurs, fix the error and recompile.



Be aware that the compiler generates messages as they are detected. Because C and C++ don't force any restrictions on placing statements on a line of text, the true cause of the error might occur one or more lines before or after the line number specified in the error message.

Warnings indicate that conditions that are suspicious but legitimate exist, or that machine-dependent constructs exist in your source files. Warnings do not stop compilation.

Warnings are issued as a result of a variety of conditions, such as:

Table A-2
*Warning
descriptions*

Warning	Description
ANSI violations	Warn you of code that is acceptable to Paradigm C++ (because of C++ code or Paradigm C++ extensions), but is not in the ANSI definition of C.
Frequent warnings	Alert you to common programming mistakes. These warning messages point out conditions that are not in violation of the Paradigm C++ language but can yield the wrong result.
Less frequent warnings	Alert you to less common programming mistakes. These warning messages point out conditions that are not in violation of the Paradigm C++ language but can yield the wrong result.
Portability warnings These	Alert you to possible problems with porting your code to other compilers. usually apply to Paradigm C++ extensions.
C++ warnings	Warn you of errors you've made in your C++ code. They might be due to obsolete items or incorrect syntax.

Run-time errors and warnings

Run-time errors occur after the program has successfully compiled and is running. Run-time errors are usually caused by logic errors in your program code. If you receive a run-time error, you must fix the error in your source code and recompile the program for the fix to take effect.

Linker errors and warnings

As a rule, linker errors do not stop the linker or cause .AXE or .MAP files to be deleted. When such errors happens, don't try to execute the .AXE file. Fix the error and relink.

A fatal link error, however, stops the linker immediately. In such a case, the .AXE file is deleted. All Linker errors are treated as fatal errors if you are compiling from the Paradigm C++ IDE.

Linker warnings point out conditions that you should fix. When warnings occur, .AXE and .MAP files are still created.

Librarian errors and warnings

Librarian errors and warnings occur when there is a problem with files or extended dictionaries, when memory runs low, or when there are problems as libraries are accessed.

Paradigm C++ debugger messages

Paradigm C++ debugger messages are generated by the integrated debugger and appear under the Run-time tab of the Message window. Many of these messages relate to options not set properly in the Paradigm C++ IDE screens.

ObjectScripting error messages

ObjectScripting error messages are messages that result from running scripts in the Paradigm C++ IDE. They appear under the Script tab in the Message window.

Message formats

Messages are displayed with the message class first, followed by the source file name and line number where the error was detected, and finally with the text of the message itself.

Many of the messages appear in the Message view. For those messages, context-sensitive help is available. Point to the message and press *F1* to display the message description.



If you working from the command line or want to look up information on an error message, refer to the alphabetical list of error and warning messages in "Alphabetical list of messages" later in this chapter. A listing is also available in the online Help Book Shelf under "Paradigm C++ error and warnings". The Book Shelf can be accessed by choosing Help|Keyboard and clicking on the Book Shelf menu tab.

Symbols in messages

Some messages include a symbol (such as a variable, file name, or module) that is taken from your program. In the following example, 'filename' will be replaced by the file causing the problem:

```
Error opening 'filename' for output
```

The following table describes the meaning of symbols in error and warning messages.

Table A-3
*Symbols in error
messages*

Symbol	Meaning
address	A hexadecimal number indicating the address where the error occurred
argument	An argument
base	The name of a base element such as a base class
class	A class name
constructor	The name of a constructor such as a class constructor
filename	A file name (with or without extension)
function	A function name
group	A group name
identifier	An identifier (variable name or other)
language	The name of a programming language
len	An actual number
macroname	The name of a macro
member	The name of a data member or member function
message	A message string
module	A module name
name	Any type of name
num	An actual number
operator	The symbol for an operator such as ++
option	An option
parameter	A parameter name
path	A path name

reason	Reason given in message
segment	A segment name
size	An actual number
specifier	A type specifier
symbol	A symbol name
type	A type name
variable	A program variable

Some messages begin with a symbol name such as the following:

```
'filename' not found
```

These messages are listed alphabetically using the name of the symbol. The above message would be filed under **f**.

Alphabetical list of Paradigm C++ debugger messages

To find this error message, look under the alphabetized listing of "function."

Messages are listed in ASCII alphabetic order. Messages beginning with symbols come first, then messages beginning with numbers, and then messages beginning with letters of the alphabet. Messages that begin with symbols are alphabetized by the type of the symbols. For example, you might receive the following error message if you incorrectly declared your function *my_func*:

```
my_func must be declared with no parameters
```

Bad line number 'linenumber'

You tried to add a source breakpoint at a specific line number but you typed an invalid line number. Use the Paradigm C++ IDE and correct the line number in the Add Breakpoint dialog box. Breakpoints must be set on executable lines of code.

Can't convert 'string' [which evaluates to 'result'] to an address

The debugger dialog was expecting a memory address as input and it couldn't interpret the user input as a valid address.

Can't debug during asynchronous compile

While compiling code with the Environment|Process Control|Asynchronous option set, you tried to issue a debugger command. Because the compiler is not re-entrant and the debugger and browser use the compiler code, you cannot debug or browse while an asynchronous (background) compile is taking place.

Can't evaluate 'expression:' 'reason'

The expression you tried to evaluate did not return a valid value. This error will be given any time invalid input is entered in a debugger dialog and there is no more information about the error. Every debugger dialog uses the debugger's evaluator to validate and interpret user input.

Can't inspect 'itemname'

You specified an invalid item for inspection.

Can't navigate to address 0

You are trying to bring up a source view on an address that evaluates to 0.

Can't run to 'filename', line 'linenumber'

You tried to run the specified line of the specified file. Either the file does not exist or there is no executable code associated with the line.'

Disable Group checked but no value entered

You checked the Disable Group check box, but forgot to specify a group name.

Enable Group checked but no value entered

You checked the Enable Group check box, but forgot to specify a group name.

Ensuring executable is up to date

Paradigm C++ is checking to be sure that the executable file is up to date, recompiling, if necessary.

Error: File not specified

You forgot to specify a filename in the Run To dialog.

Error: Line not specified

You forgot to specify a line number in the Run To dialog.

Error trying to change value

You tried to change a value of an object being inspected, but the debugger was unable to change the value.

Eval Expr checked but no value entered

You checked the Eval Expr check box, but forgot to specify an expression.

Expr True check but no value entered

You checked the Expr True check box, but forgot to provide an expression.

File 'filename' does not exist

You tried to bring up a source view on an address, and the associated file does not exist. This problem can usually be fixed by setting the appropriate source path on the debugger option page.

File 'filename' does not exist (trying to load it anyway...)

The debugger tried to load an executable that does not exist. Check to make sure that the executable exists and that the path to the executable was correctly specified.

File name not specified

You tried to add a source breakpoint using the Paradigm C++ IDE, but you omitted a file name. Enter the name of the file into which you want to insert the breakpoint in the Add Breakpoint dialog box.

Function call terminated by unhandled exception 'value' at address 'addr'

This message is emitted when an expression you are evaluating while debugging includes a function call that terminates with an unhandled exception. For example, if in the debugger's evaluate dialog, you request an evaluation of the expression `f○○() + 1` and the execution of the function `f○○()` causes a GP fault, this evaluation produces the above error message.

You may also see this message in the Watches window because it also displays the results of evaluating an expression.

Group name not specified

You tried to set breakpoint options in the Breakpoint Condition/Action Options dialog box but forgot to specify a group name.

Invalid Pass Count value entered

The Pass Count value you gave was invalid. Valid values for Pass Count are from 0 to 4294967295.

Invalid pathname for executable

The debugger was unable to find the executable you tried to load.

Invalid process id

You specified a process ID that does not match the ID of any active process.

Loading: 'programname'

The debugger is loading the specified program.

Log Expr checked but no value entered

You checked the Log Expr check box, but forgot to specify an expression.

Log Msg checked but no value entered

You checked the Log Msg check box, but forgot to specify a message.

Make failed

The make spawned by the debugger to try to bring the current target up to date failed. Check the Build Time tab in the Message view to see the reason for the failure.

Make the modified code?

You had a process loaded in integrated debugger and then you modified the source code for the process. You should probably build the new code instead of continuing to debug the old executable.

No expression specified

You forgot to specify an expression in the Add Watch dialog

No file corresponds to this item

You tried to bring up a source view on an address, and there is no source file for the address.

No file line specified

You tried to add a Source breakpoint using the Paradigm C++ IDE, but did not include the line number. Specify the line in the file where you want the breakpoint to occur in the Add Breakpoint dialog box.

No line corresponds to this item

You tried to bring up a source view on an address, and there is no line number for the address.

No module name specified

You tried to add a module breakpoint using the Paradigm C++ IDE, but you omitted the module name. Specify the module name where you want to insert the breakpoint in the Add Breakpoint dialog box.

No module specified

You tried to add an Address breakpoint using the Paradigm C++ IDE, but you omitted the module. Specify the module where you want to insert the breakpoint in the Add Breakpoint dialog box.

No object specified

You tried to add an Address breakpoint using the Paradigm C++ IDE, but you omitted the object. Specify the name of the object into which you want to insert the breakpoint in the Add Breakpoint dialog box.

No offset specified

You tried to add an Address breakpoint using the Paradigm C++ IDE, but you omitted the offset that indicates where you want to insert the breakpoint. Specify the offset in the Add Breakpoint dialog box.

No process selected

You pressed the Attach button on the debugger's Attach dialog when there was no process selected in the process list.

No process to load

You left the Program Name field blank on the Load Program dialog.

No process to reset

You tried to reset a process but there was no process running.

No process to stop

You tried to pause a process but there was no process running.

No process to terminate

You tried to terminate processes but there was no process running at the time.

No type specified

You tried to add a C++ exception breakpoint using the Paradigm C++ IDE. You must specify a type in the Add Breakpoint dialog box to set this type of breakpoint.

No watch address specified

You specified a data watch breakpoint using the Paradigm C++ IDE, but you omitted the watch address. You need to specify both a memory address and the number of bytes to watch.

No watch length specified

You specified a data watch breakpoint using the Paradigm C++ IDE, but you omitted the watch length. You need to specify both a memory address and the number of bytes to watch.

Not all breakpoints were valid

You set breakpoints in your program but they were not all valid. Check the breakpoint view to see which breakpoints were invalid.

OS exception number not specified

You tried to add an OS exception breakpoint using the Paradigm C++ IDE. You must include an OS exception number if you want to add a breakpoint when a particular OS exception occurs. Select one of the exceptions in the list box next to the Exception # field or enter a user-defined exception number.

Pass Count checked but no value entered

You checked the Pass Count check box, but forgot to provide a pass count. You need to specify a valid pass count.

Process created: 'processname'

The process specified in the message has been created.

Process 'processname' (0x%X) is already being debugged

You tried to attach to a process that is already being debugged.

Process 'processname' (0x%X) is Paradigm C++

You tried to attach to the Paradigm C++ IDE. This is not allowed. Specify another process.

Process Stopped: 'processname'

The process specified in the message was stopped.

Process terminated: 'programname'

The specified process has been terminated.

Resetting

The process is being reset to its initial condition.

Running

The process is running.

Stopping

The process is stopping.

Terminating

The process is terminating.

The expression cannot be modified

This is an integrated debugger error. You entered an expression in the Evaluator dialog box and clicked on Modify but the expression cannot be modified.

The expression you entered could not be evaluated

This is an integrated debugger error. The integrated debugger could not interpret the expression you entered in the Evaluator dialog box.

There is no code for 'file', line 'linenumber'

You tried to view the disassembly for the given line of source code. The specified line of the file has no code associated with it.

There is no expression to evaluate

This is an integrated debugger error. You forgot to enter an expression in the Evaluator dialog box.

There is no expression to evaluate, and no process is loaded

This is an integrated debugger error. You forgot to enter an expression in the Evaluator dialog box and no program is loaded.

This operation not supported for 16 bit executables

You tried to use a command (such as Reset or Pause) in the integrated debugger while the project was set to produce a 16-bit executable. The integrated debugger does not support 16-bit executables except to run or terminate them.

Alphabetical list of Compiler messages

To find this error message, look under the alphabetized listing of "function."

Messages are listed in ASCII alphabetic order. Messages beginning with symbols come first, then messages beginning with numbers, and then messages beginning with letters of the alphabet. Messages that begin with symbols are alphabetized by the type of the symbols. For example, you might receive the following error message if you incorrectly declared your function *my_func*:

```
my_func must be declared with no parameters
```

Cannot access an inactive scope

You have tried to evaluate or inspect a variable local to a function that is currently not active. (This is an integrated debugger expression evaluation message.)

Cannot evaluate function call

The error message is issued if someone tries to explicitly construct an object or call a virtual function.

In integrated debugger expression evaluation, calls to certain functions (including implicit conversion functions, constructors, destructors, overloaded operators, and inline functions) are not supported.

Cannot take address of member function 'function'

An expression takes the address of a class member function, but this member function was not found in the program being debugged. The evaluator issues this message.

Invalid 'expression' in scope override

The evaluator issues this message when there is an error in a scope override in an expression you are watching or inspecting. You can specify a symbol table, a compilation unit, a source file name, etc. as the scope of the expression, and the message will appear whenever the compiler cannot access the symbol table, compilation unit, or whatever.

Invalid function call

A requested function call failed because the function is not available in the program, a parameter cannot be evaluated, and so on. The evaluator issues this message.

Missing 'identifier' in scope override

The syntax of a scope override is somehow incomplete. The evaluator issues this message.

'new' and 'delete' not supported

The integrated debugger does not support the evaluation of the new and delete operators.

No type information

The integrated debugger has no type information for this variable. Ensure that you've compiled the module with debug information. If it has, the module may have been compiled by another compiler or assembler.

Not a valid expression format type

Invalid format specifier following expression in the debug evaluate or watch window. A valid format specifier is an optional repeat value followed by a format character (c, d, f[n], h, x, m, p, r, or s).

Overloaded function resolution not supported

In integrated debugger expression evaluation, resolution of overloaded functions or operators is not supported, not even to take an address.

Repeat count needs an lvalue

The expression before the comma (,) in the Watch or Evaluate window must be an accessible region of storage. For example, expressions like this one are not valid:

```
i++, 10d  
x = y, 10m
```

String literal not allowed in this context

This error message is issued by the evaluator when a string literal appears in a context other than a function call.

The function 'function' is not available

You tried to call a function that is known to the evaluator, but which was not present in the program being debugged for example, an inline function.

- #
- !elif 186
- !else 186
- !endif 186
- !error 185
- !if 186
- !ifdef 186
- !ifndef 186
- !include 187
- !message 187
- !undef 188
- #if 141
- #ifdef 141
- \$ENV() 85
- \$INHERIT 85
- .autodepend 185
- .path.ext 187
- .precious 188
- .suffixes 188
- /(slash)
 - 16-bit linker options 86
 - 32-bit linker options 89
 - command-line options 109
 - Directory options 83
 - General options 92
 - Librarian options 85
 - Map options 94
 - Source Directories options 82
 - Warnings options 96
- ;(semi-colon) 208
- _ (underscores) 63, 77
 - __cdecl 53, 63
 - __far 61, 62
 - __fastcall 53, 63
 - __fastthis 74
 - __huge 67
 - __pascal 53, 63
 - __stdcall 63
 - __BSS 60
- 1 compiler option 57, 212
- 16- and 32-bit command-line options 109
- 16- and 32-bit compiler options 109
- 16-bit command-line options 111
- 16-bit compiler options 53, 111
 - calling conventions 53
 - memory model 54
 - processor 57
 - segment names code 59
 - segment names data 60
 - segment names far data 61
- 16-bit linker options 86
 - calling conventions 87
 - enabling 32-bit processing 86
 - initializing segments 87
 - names tables 87
 - nonresident names table 86
 - segment alignment 87
- 16-bit memory management 241
- 16-bit optimization 88, 104
- 2 compiler option 57
- 3 compiler option 57, 64
- 32-bit command-line options 109, 112
- 32-bit compiler options 62, 109
 - calling convention 63
 - processor 64
- 32-bit instruction set 64
- 32-bit linker options 89
 - committed heap size 90
 - committed stack size 90
 - file alignment 90
 - image base address 91
 - image is based 91
 - importing by ordinal 89
 - incremental linker 92
 - linker errors 91
 - object alignment 91
 - reserved heap size 91
 - reserved stack size 92
 - verbose 92
- 32-bit optimization options 106
- 32-bit, enabling 86
- 32RTM.EXE 170
- 4 compiler option 57, 64
- 5 compiler option 57, 64
- 8086
 - processors 57
 - registers 241
- 80x86 processors 57, 64
 - instruction opcodes 211
 - registers 208
- 80x87 coprocessors 235
 - emulating 235
 - registers 237
- 87 environment variable 236
- B compiler option 211
- i486 instructions 57, 64

A

- .autodepend 185
- A compile options 78
- a compiler options 58
- Add Node command 36
- Add Target dialog box 38
- addresses 56, 58, 91
 - map files 95
- Advanced Options dialog box 35
- AK compiler option 78
- algorithms 83
- aliases 104
- alignment 58
 - byte 58
 - double word 59
 - file 90
 - object 91
 - quad word 59
 - segments 87
 - word 59
- alloc.h 219
- Allocate Enums As Ints option 73
- allocation 73, 105
- alphabetical listings
 - error messages and warnings 271, 276
- ancestors 126
- ANSI 78, 98
- arguments 228, 233
 - passing 53, 63, 67
- arithmetic 238
- arrays
 - project options 56, 102
- asm keyword 207
 - nesting 208
- assembly language 207
 - calling conventions 208
 - comments 208
 - directives 213
 - floating-point emulation 211
 - instructions 207
 - jump instructions 210, 213
 - new lines 207
 - opcodes 211
 - operands 207
 - references 208
 - registers 208
 - repeat prefixes 212
 - size instruction 209
 - statements 207
 - C symbols 207
 - string instructions 212
 - structures 209

- syntax 207
 - variable offsets 209
- assert 220
- assert.h 220
- assignment 87, 101
- AT compiler option 78
- AU compiler option 78
- autodependencies 76, 97
- Automatic Far Data option 54

B

- _BSS 60
- b compiler option 73
- background compile 271
- base addresses 91
- bcd, binary-coded decimals 238
 - converting 240
- binary-coded decimals 238
- Break make on option 97
- Breakpoint Condition/Action options dialog box 145
- breakpoints 136
 - adding 136, 143
 - conditional 137
 - customizing 142, 145
 - color 142
 - disabling/enabling 140, 145
 - groups 141
 - editing 148
 - inspecting 140
 - option sets 141
 - removing 139
 - resetting invalid 141
 - setting 135, 136, 137
 - conditional 136
 - unconditional 136
 - type 143
 - viewing 140
- Breakpoints window 136, 137, 140, 142
- browser 125
 - customizing 128
 - starting 125
 - using menu commands 125
 - views 125
- Browser options 79
- Browser Reference Information in OBJs option 79
- browsing
 - class inspection 127
 - filters and letter symbols 127
 - global symbols 126
 - objects 126
 - references 127

- symbol declaration 127
- symbols 126
- Build All command 51
- Build attributes option 65
- Build Node command 51
- building
 - applications 97
 - libraries 191
- builds 51, 65, 84, 173
- BUILTINS.MAK 173, 174
- byte alignment 58

C

- __cdecl 53, 63
- C calling conventions 53, 63
- C compile option 78
- C++ coding, inefficient 99
- C++ options 65
 - compatibility 65
 - exception handling 68
 - general 70
 - member pointers 70
 - templates 71
 - virtual tables 71
- cache hit optimizations 106
- calculations 76, 105
- call stack 80
- Call Stack window 163
- calling conventions 66, 233
 - __fastthis 74
 - compiling options 53, 63, 74
 - optimizing 87, 103
 - Pascal 53, 63, 67
- case sensitivity 85
 - exports and imports 92
 - link 93
- catch 199
- C-based structured exceptions 204
- character conversion macros 220
- character types 66, 75
- child nodes 32
- Class Inspection window 127
- class member functions 135
- classes 257
 - compiling options 65, 66, 67
 - declarations 223
 - empty base classes 70
- Classes command 126
- code
 - classes 59
 - elimination 105
 - external 135

- groups 59
- inefficient coding 100
- motion, optimizing 105
- page alignment 87
- searching 125
- segments 39, 54, 55, 57, 59, 62
 - packing 93, 94
 - unreachable 100
- code generation 79, 94
 - compiler 57, 72, 73, 77
 - optimization 104, 105
- code pages 90
- color customization
 - syntax highlighting 142
- COMDEFs 55, 77
- command-line compilers 166
- command-line options 109, 166
 - 16- and 32-bit 109
 - 16-bit 111
 - 32-bit 112
 - by function 118
 - compiler 112
 - exception handling 203
 - MAKE 175
 - object search paths 109
 - PLIB 191
 - PLINK 167
- command-line tools
 - running 170
- comment records, purging 86
- comments, nested 78
- communal variables 54, 77
- compact memory models 54, 56, 57
- compatibility 65
- Compile command 51
- compiler errors and warnings 268
 - declarations 277
 - evaluating expressions 276, 277
 - function calls 272, 276, 277
 - lvalue 277
 - modules 274
 - watch address 274
- compiler options 72
 - assembly 210
 - code generation 73
 - compiler output 76
 - debugging 79
 - defines 72
 - precompiled headers 81
 - source 77
- compiler output options 76
 - autodependencies 76
 - generating code 77

- generating underscores 77
- compilers 72, 166
 - 32-bit command-line options 111, 112
 - command-line options 112
 - message options 98
 - project options 52, 53, 62, 65, 72, 101
 - stopping 101
- compile-time errors
 - fixing 52, 129
- compiling 51, 56, 173, 224
 - optimizing 169
 - with symbol tables 130
- complex numbers 238
- conditional breakpoints 137
- configuration files 166
- constants 104
- constructors 66, 203
- context-sensitive help 28
- conversions 230, 232
- converting old projects 39
- copy propagation 104
- CPU instruction sets 57, 64
- CPU window 156
 - Disassembly pane 157
 - Flags pane 162
 - Memory Dump pane 159
 - Registers pane 161
 - Stack pane 160
- ctype.h 220
- customizing *See* Environment options
- customizing the browser 128

D

- _defs.h 233
- D compile option 72
- d compiler option 73
- data
 - alignment 58
 - inspecting range 153
 - members 68
 - objects 54
 - segments 39, 54, 55, 57, 60, 61
 - structures 148
 - value 148
- dc compiler option 57
- debug options
 - environment 142
 - syntax highlighting 142
- debugger 20, 129
 - adding breakpoints 136
 - compile-time errors 52
 - conditional breakpoints 137

- customizing 131
- debug information 79, 93, 130
- evaluating expressions 154
- external code 135
- fixing errors 52, 148
 - logic 129
 - run-time 129
- inspecting code 140, 153
- messages and warnings 267, 269
- modifying variables 155
- optimizing 105, 108
- options
 - pausing a program 135
 - program arguments 131
 - restarting a program 135
 - terminating a program 135
- program execution 131
- running programs 132
- setting watches 148
- SpeedButtons 21
- starting a session 130
- stepping 133, 163
- target connection 19
 - stand-alone 19
- viewing errors 52
- debugging macro, assert 220
- debugging options 79
 - browser 79
 - debug information in OBJs 79
 - line numbers 79
 - out-of-line inline functions 80
 - stack frame 80
 - test stack overflow 80
- declarations
 - classes 223
 - errors 277
- default libraries, linker options 93
- defining
 - macros 73, 182
 - variables 77
- dependencies 76
 - checking 97
- derived classes 65, 66
- descendants 126
- desktop
 - speedbar options 25
- destructors 69, 203
- detailed segment maps 95
- DGROUP 60
- dictionaries 86
 - extended 192, 193
- directives
 - assembly 213

- MAKE 184, 186, 188
- directories 82, 109, 166
 - options 82
 - entering directory names 84
 - file search algorithms 83
 - output 84
 - source 82
- directory names, entering 84
- disable all, optimization option 108
- disabling messages and warnings 98
- Disassembly pane 157
 - SpeedMenu 158
- display warnings 98, 101
- DLLs 86
- DOS applications
 - compiling options 54, 56
- dos.h 221
- double 232
- double word alignment 59
- Dump pane 159, 160
 - SpeedMenu 160
- duplicate strings 74
- duplicate symbols, linker warning 96
- dynamic mode 153
- dynamic-link libraries 86

E

- !elif 186
- !else 186
- !endif 186
- !error 185
- \$ENV() 85
- Edit window 11, 14, 52, 136, 137
- editing code 140
- editor 14
 - options 22
- EDPML.SWP 170
- embedded.h 221
- enumeration types 73
- Environment options 22
 - browser 128
 - debugger 131
 - Editor 23
 - Preferences 27
 - project views 39
 - SpeedBar 25
 - Syntax highlighting 24
- environmental parameters 224
- errno.h 222
- error codes 222
- error messages 269
 - alphabetical listings 271, 276

- categories 267
- compiler 268
- fatal errors 267
- informational 268
- librarian 269
- linker 269
- ObjectScripting 270
- run-time 269
- warnings 268
- error-handling mechanism 197
- errors 267
 - 32-bit linker 91
 - C++ 100
 - compile-time 52
 - declaration syntax 78
 - fixing 52, 129
 - header file 222
 - linker 96
 - linker errors 91
 - logic 129
 - messages options 98
 - potenial errors 101
 - run-time 129
 - stop after... 101
 - viewing 52
- Eval Expr 272
- evaluating expressions 154
- exception handling 227, 233
 - options 68
 - routines 68
- exceptions 197
 - catch keyword 199
 - C-based structured 204
 - command-line options 203
 - compiling options 68
 - constructors and destructors 203
 - enabling 68
 - exception declarations 198
 - floating-point 237
 - handling 68
 - throwing exceptions 198
 - unhandled exceptions 203
- excpt.h 233
- executable (EXE) files 84, 86, 167
- execution point 132, 148
- expanding inline functions 68, 71, 80
- explicit
 - casts 70
 - libraries 83
- exporting 87, 92
- Expr True 272
- expressions
 - duplicate 102

- evaluating 154
- format specifiers 155
- optimizing 102, 104, 105
- extended dictionaries 86, 192, 193
- external code 135
- external option 71
- external references 71, 72
- external symbols 93

F

- __far 61, 62
- __fastcall 53, 63
- __fastthis 74
- f compiler option 75
- Fa compile option 55
- far
 - calls 87
 - classes 61, 62, 67
 - data compatibility 55
 - data segments 54, 55, 61, 62
 - declaring functions 253
 - declaring objects 253
 - declaring pointers 254
 - Far data threshold 55
 - initialized data groups 61
 - objects 61, 62
 - pointers 246
 - uninitialized data groups 62
 - virtual tables 55, 62
- FAR_BSS 62
- FAR_BSS class 55
- FAR_DATA 61
- FAR_DATA class 55
- fastcall parameter-passing 53, 63
- fastthis calling convention 74
- fatal errors 98, 267, 269
- Fb compile option 55
- Fc compiler option 77
- fcntl.h 222
- Ff compile option 54, 55
- ff compiler option 75
- file alignment 90
- file extensions
 - .DEF 92
 - .DLL 84, 86, 87
 - .EXE 84, 86, 167
 - .LIB 82, 83, 85, 86, 96
 - .LST 86
 - .MAP 84
 - .OBJ 84, 85, 86, 94, 109, 130
 - .PDL 47
 - .ROM 130

- syntax highlighting 24
- file names 81
- file search algorithms 83
- files 166
 - 32RTM.EXE 170
 - BUILTINS.MAK 174, 176
 - creating 14
 - include 82
 - MAKE.EXE 173, 176
 - MAKESWAP.EXE 170
 - PLIB.EXE 191
 - PLINK.CFG 168
 - TOUCH.EXE 174
- file-sharing 228
- filling and padding 261
- filters and letter symbols 127
- Filters matrix 127
- Finder 15
- fixing errors 52, 129
- Flags pane 162
 - SpeedMenu 163
- flags registers 243
- float 232
- float.h 223
- floating point
 - calculations 75
 - emulation (inline assembler) 211
- I/O 235
 - code 236
 - exceptions 237
 - fast option 236
 - options 75
 - routines 223
- Flyby Help Hints 25
- Fm compiler option 54
- for statements 66
- format specifiers
 - expressions 155
- fp compiler option 75
- Fs compiler option 54
- function boundaries, optimizing 107
- function calls 103, 163, 233, 272, 276
 - compiler error 276
 - compiling options 53, 63
 - errors 277
- functions
 - class member 135
 - inline 68, 71, 79, 80, 103

G

- G compile option 108
- g compiler option 101

- General linker options 92
 - case-sensitive link 93
 - code pack size 93
 - debug information 93
 - default libraries 93
 - exports and imports 92
 - pack code segments 94
 - subsystem version 94
- general warnings 99
- Generate COMDEFs option 77
- generating code 79, 94
 - compiler options 57, 72, 73, 77
 - optimization 105
- generating underscores
 - compiler options 77
- generic.h 223
- global
 - definitions 71
 - registers 105
 - symbols 126
 - variables
 - project options 54, 58, 77
- globals command 126
- glyphs
 - Project Manager 32
- groups 273
 - breakpoint 141

H

- __huge 67
- H compile option 81
- h compiler option 56
- H"xxx" compiler option 82
- H=filename compiler option 81
- Hc compiler option 81
- header files 83, 215
 - _defs.h 233
 - _nfile.h 233
 - _null.h 234
 - alloc.h 219
 - assert.h 220
 - ctype.h 220
 - dos.h 221
 - embedded.h 221
 - errno.h 222
 - excpt.h 233
 - fcntl.h 222
 - float.h 223
 - generic.h 223
 - io.h 223
 - iomanip.h 224
 - limits.h 224

- malloc.h 224
- math.h 225
- mem.h 226
- memory.h 227
- new.h 227
- precompiled 81
- process.h 227
- search.h 227
- setjmp.h 227
- share.h 228
- signal.h 228
- stdarg.h 228
- stddef.h 229
- stdio.h 229
- stdiostr.h 230
- stdlib.h 230
- string.h 231
- sys\locking.h 231
- sys\type.h 231
- time.h 232
- values.h 232
- varargs.h 233
- heap 90, 91
- heap size 90, 91
 - committed 90
 - reserved 91
- Help 28
 - contacting Paradigm 30
 - context-sensitive help 29
 - displaying Contents 29
 - Help files 28
 - index 29
 - keyword searches 29
 - printing topics 29
 - SpeedMenus 30
- hidden
 - members 66
 - pointers 65, 67
- Hu compiler option 81
- huge
 - arrays 56
 - declaring pointers 254
 - memory models 54, 56
 - pointers 246

I

- !if 186
- !ifdef 186
- !ifndef 186
- !include 187
- #if 141
- #ifdef 141

- \$INHERIT 85
- i486 instructions 64
- I compile option 82, 83
- i compiler option 77
- I/O 235
 - exceptions 237
 - formatting 260
 - manipulators 224
 - of user defined types 263
 - routines 223, 229
 - simple file 263
- i486 instructions 57
- i586 instructions 57
- Identifier length option 77
- identifiers 77
 - Pascal 77
- image base addresses 91
- implicit libraries 83
- importing 92
- importing by ordinal, linker option 89
- include files 83, 215
- incremental linker, linker option 92
- induction variables 102
- informational messages 267, 268
- inheritance 67, 70
- initialization 60, 100
 - segments 87
- initialized data 60
- inline
 - #pragma directive 210
 - assembly 207
 - functions 68, 71, 79, 80, 103
 - statements 210
- input 223, 257, 262
- inspecting 127
 - breakpoints 140
 - code 153
 - data range 153
 - error 271
 - expressions 153
 - local variables 153
- Inspector window 153
 - changing values 153
- installation 11
- instructions
 - Pentium 57, 64
 - project options 57, 64
 - string move 106
- integral quantities ranges 224
- integrated debugger 129
 - adding breakpoints 136
 - conditional breakpoints 137
 - customizing 131

- error messages 269
- errors 52, 129
- evaluating expressions 154
- inspecting code 140, 153
- messages and warnings 267
- modifying variables 155
- optimizing 105, 108
- program execution 131
- running programs 132
- setting watches 148, 149
- starting 131
- stepping 133, 163
- Intel compiler 106
- Intel optimizing compiler 62
- intrinsic functions 103
- invalid breakpoints 141
- invariant code 105
- io.h 223
- iomanip.h 224
- ios class 258
- iostream classes 257
 - ios 258
 - streambuf 257
- iostream library 257

J

- j compiler option 101
- Jg compiler option 71
- Jgd compiler option 71
- Jgx compiler option 71
- jump optimization 105

K

- K compile option 75
- k compiler option 80
- K2 compiler option 66
- Kernighan and Ritchie 78, 229

L

- L compile option 82
- Language compliance option 78
- large memory models 54, 56, 57
- Librarian messages 269
- Librarian options 85
 - case-sensitive library 85
 - comment records 86
 - dictionaries 86
 - list files 86
 - page size 86
- libraries 83
 - case-sensitive 85
 - creating 191

- default libraries 93
- dynamic-link 86
- managing 191
- project options 82, 86, 96
- library files 82, 83, 96, 255
- library functions 222
- limits.h 224
- line numbers 79
 - including 94
- linker errors 91
- Linker messages 269
- Linker options 86
 - 16-bit programs 86
 - 32-bit programs 89
 - general 92
 - map files 94
 - warnings 96
- linkers 86
 - 16-bit command-line options 111
 - command-line options 118
 - project options 86, 92, 94, 96
- linking 86, 167
 - command-line syntax 167
 - large applications 192
 - mixed modules 255
 - optimizing 86, 94, 168
- list files 86
- literal strings 57, 74
- local virtual tables 72
- locking mode parameter 231
- Log Expr 273
- Log Msg 273
- logic errors
 - fixing 129
- longjmp 227
- loops 66, 102, 105, 106
- low-level I/O routines 223
- lvalue
 - errors 277

M

- !message 187
- Machine Stack pane 160
 - SpeedMenu 161
- macros 183, 229
 - \$INHERIT and \$ENV() 84, 85
 - defining 72, 73
 - MAKE 182, 183, 184, 188, 189
- MAKE 97, 173
 - command operators 181
 - command prefixes 180
 - command syntax 180

- command-line options 175, 176
- defaults 173, 174, 176, 183
- directives 184, 186, 188
- macros 182
 - defaults 183
 - defining 182
 - in directives 188
 - modifying default 184
 - null 189
 - string substitutions 183
- NMAKE compatibility 176
- project options 97
- rules 178, 179
- TOUCH 174
- Make All command 51
- Make Node command 51
- Make options 97
 - autodependencies 97
 - Break make on 97
 - new node path 98
- makefiles 177
 - response files 181
- MAKESWAP 170
- malloc.h 224
- mangled names 55, 96
- manifest constants 72
- manipulators 224, 260
- map files 84, 95
 - linker options 94, 95, 96
- math
 - complex classes 238
 - error handlers 225
 - floating point 235
 - math.h 225
- mc compiler option 56
- medium memory models 54, 56
- mem.h 226
- member functions 135
- member pointers 70
 - honor precision 70
 - options 70
 - representation 70
- memory 86, 219, 226, 227
 - running out of 241
- Memory Dump pane 159
 - SpeedMenu 160
- memory functions 103
- memory management
 - 16-bit 241
 - functions 219, 224
- memory manipulation functions 226, 227, 231
- Memory Model options 54
 - compiling segments 54

- far data 54
- far data compatibility 55
- far data threshold 55
- huge pointers 56
- models 56
- page alignment 55
- stack and data segments 54
- strings 57
- virtual tables 55
- memory models 241, 247
 - mixed-model programming 252
- memory segmentation 244
- memory.h 227
- Menu Bar 11
 - command descriptions 12
- Message window 11, 52, 99
- messages 98
 - disabling 98
 - displaying 98, 267
 - project options 98
- Messages options 98
 - ANSI violations 98
 - display warnings 98
 - general 99
 - inefficient C++ coding 99
 - inefficient coding 100
 - obsolete C++ 100
 - portability 100
 - potential C++ errors 100
 - potential errors 101
 - stop after... errors 101
 - stop after... warnings 101
 - user-defined warnings 99
- mh compiler option 56
- mixed-model programming 255
- ml compiler option 56
- mm compiler option 56
- mm! compiler option 56
- module definition files 92
- modules 53, 62, 274
 - purging comment records 86
- ms compiler option 56
- ms! compiler option 56
- mt compiler option 56
- multiple directories 84
- multi-target projects 37

N

- _nfile.h 233
- _null.h 234
- N compile option 80
- n compiler option 84

- name mangling 55, 66, 96
- name tables 86, 87
- near
 - declaring functions 253
 - declaring pointers 254
 - pointers 246
- nested comments 78
- nested templates 101
- New Target command 17, 38
- new.h 227
- NMAKE 176
- Node attributes dialog box 37
- node path, Make option 98
- nodes 40, 65
 - adding 33, 36
 - building 51
 - changing attributes 38
 - copying 38
 - default 35
 - deleting 36
 - Make Node command 51
 - options 37, 39, 49, 50
- nonresident name tables 86, 87
- nonstatic data members 68
- normalizing huge pointers 56
- null 189
- numerical types 238

O

- OI compile option 106
- OM compile option 106
- OS compile option 106
- O compile option 105
- O1 compiler option 108
- O2 compiler option 108
- Oa compiler option 104
- Ob compiler option 105
- object alignment options 91
- object files 109, 130
 - project options 76, 79, 82, 84, 94, 96
 - searching 109
- object hierarchies 126
- object search paths 109
- objects 54, 61, 62
 - sharing 55
- ObjectScripting messages 270
- obsolete C++ 100
- Oc compiler option 102
- Od compiler option 108
- Oe compiler option 105
- offsets 56
- Og compiler option 102

- Oi compiler option 103
- Ol compiler option 106
- Om compiler option 105
- online help 28
- Op compiler option 104
- opcodes 211
- opening projects 36
- operators 101
- Optimization options 101, 104
 - 16- and 32-bit 102, 105
 - 16-bit 88, 104
 - 32-bit 106
 - common subexpression 102
 - copy propagation 104
 - dead code elimination 105
 - disable all 108
 - general settings 108
 - induction variables 102
 - inline intrinsic functions 103
 - invariant code motion 105
 - jump optimization 105
 - loop optimization 106
 - pointer aliasing 104
 - project options 101
 - size 105
 - suppress loads 106
- optimizing 101
 - debugger 105, 108
 - expressions 102, 104, 105
 - far call to near 87
 - jumps 105
 - size 104
 - statements 101, 105
- ordinal numbers 89, 92
- Os compiler option 108
- Ot compiler option 108
- out-of-line inline functions 72, 80
- output 223, 235, 257, 259
 - directories 84
 - files 84
- Ov compiler option 102
- overrides 65
- Ox compiler option 108

P

- #pragma directives 210
- .path.ext 187
- .precious 188
- __pascal 53, 63
- p compiler option 53
- p compiler options 63
- packing code segments 93, 94

- page alignment 55, 87, 90
- page size 86
- Paradigm C++ IDE messages 269
- Paradigm C++ messages 267
- Paradigm C++ tools overview 170
- Paradigm extensions 78
- Paradigm optimizing compiler 62
- Paradigm Systems, contacting 30
- parameterized manipulators 224
- parameters 53, 228, 233
 - passing 63, 67
- parent nodes 32
- Pascal 63
 - calling conventions 67
 - identifiers 77
- PASM (inline assembler) 211
- pass count 146
 - error 273
- PCC.EXE 109
- PCC32.EXE 109
- PCC32i.EXE 109
- PDL files 47
- Pentium instruction scheduling 106
- Pentium instructions 57, 64
- Pentium option 64
- PLIB 191
 - /C option 192
 - /E option 192
 - /P option 193
 - command-line options 191
 - error messages 267
 - examples 194
 - operation list 193
 - project options 85
 - response files 193
- PLIB.EXE 85
- PLINK 169
 - command-line options 111
 - command-line syntax 167
 - error messages 267
 - optimizing 192
- PLINK and PLINK32 86
 - 16-bit options 86
 - 32-bit options 89
 - command-line options 112, 118
 - general options 92
 - map files 94
 - warnings 96
- PLINK.CFG 168
- po compiler option 74
- pointer aliasing, optimization 104
- pointers 245
 - compiling options 55, 56, 65, 67, 68, 70, 104

- declaring 254
- far 246
- huge 246
- near 246
- segment 252
- portability 100
- precision 101
- precompiled headers 81, 216
 - cache 81
 - files 81
 - header name 81
 - terminating 82
- preprocessing 72
- Print mangled names 96
- process.h 227
- Processor options 57, 64
 - 16-bit compiler 57
 - 32-bit compiler 64
 - 32-bit instruction set 64
 - alignment 58
 - instructions 57
- project management 31
- Project Manager 32
 - nodes 33
- Project options 52
 - 16-bit compiler 53, 54, 59, 60, 61
 - 32-bit compiler 62, 64
 - build attributes 65
 - C++ 65
 - compatibility 65
 - exception handling 68
 - general options 70
 - member pointers 70
 - options 65
 - templates 71
 - virtual tables 71
- command-line options 109
 - 16- and 32-bit 109
 - 16-bit 111
 - 32-bit 112
 - by function 118
 - compiler 112
 - object search paths 109
- compiler 72
 - code generation 73
 - compiler output 76
 - debugging 79
 - defines 72
 - floating point 75
 - precompiled headers 81
 - source 77
- directories 82
 - file search algorithms 83
 - names 84
 - output 84
 - source 82
- librarian
 - case-sensitive library 85
 - comment records 86
 - dictionaries 86
 - list files 86
 - page size 86
- linker 86
 - 16-bit programs 86
 - 32-bit programs 89
 - general 92
 - map files 94
 - warnings 96
- Make 97
 - autodependencies 97
 - Break make on 97
 - new node path 98
- messages 98
 - ANSI violation 98
 - display warnings 98
 - general warnings 99
 - inefficient C++ coding 99
 - inefficient coding 100
 - obsolete C++ 100
 - portability 100
 - potential C++ errors 100, 101
 - stop after... errors 101
 - stop after... warnings 101
 - user-defined warnings 99
- optimization 101
 - 16- and 32-bit 102, 105
 - 16-bit 104
 - 32-bit 106
 - common subexpression 102
 - copy propagation 104
 - dead code elimination 105
 - general settings 108
 - induction variables 102
 - inline intrinsic functions 103
 - invariant code motion 105
 - jump optimization 105
 - loop optimization 106
 - pointer aliasing 104
 - suppress loads 106
- project tree 32
 - default nodes 35
 - navigating 33
- Project View options 39
- Project window 16
- projects 16, 31, 40, 45, 49, 173, 174
 - building files 51, 173

- compiling 51
- converting 39
- creating 17
- Make Node command 51
- multi-target 37
- setting preferences 27
- sharing tools 47
- viewing options 50
- public definitions 71, 72
- public symbols 93
 - map files 95

Q

- quad word alignment 59

R

- R compile option 79
- r compiler option 74
- raise 228
- rd compiler option 74
- redundant loads, suppressing 106
- reference nodes 38
- references 127
 - compiling options 67, 71
- register keyword 74
- register variables 74
- registers 105, 106
 - 8086 241
 - flags 243
 - general-purpose 242
 - reloading 106
 - segment 243
 - special-purpose 243
- Registers pane 161
 - SpeedMenu 161
- reloading registers 106
- repeat prefixes 212
- reserved words 78
- resident names 87
- response files 166, 169, 193
- routines
 - exception handling 68
- RT compiler option 68
- RTM.EXE 171
- RTTI 68
- run-time errors 269
 - fixing 129
- run-time support 227
- run-time type information 68

S

- .suffixes 188

- __stdcall 63
- Save command 27
- scratch registers 209
- search 227, 230
 - code 125
 - paths 83, 109, 166
- Search menu
 - classes 126
 - globals 126
- search.h 227
- segment 39
 - alignment 87
 - compiling options 54, 55, 57, 59, 60, 61
 - initializing 87
 - linker code 87
 - map files 95
 - names 59, 60, 61
 - code options 59
 - far initialized data 61
 - far uninitialized data 62
 - far virtual tables 62
 - initialized data 60
 - uninitialized data 60
 - packing code 93, 94
 - pointers 252
 - registers 243
- segments and offsets 255
- setjmp 227
- setjmp.h 227
- settings
 - optimization 108
- share.h 228
- sharing objects 55
- signal 228
- signal.h 228
- signed character types 66
- single stepping 133
- size, optimizing 104, 105, 106
- small memory models 54, 56
- sorting 227, 230
- source code 94, 125
- source directories 82
- source files 82
- Source options 77
 - identifier length 77
 - language compliance 78
 - nested comments 78
- source pools 40
 - creating 40
- speed
 - optimizing 106
- speed, optimizing 102, 103, 104, 105
- SpeedBar

- copying 27
- SpeedMenus 13
- stack 80, 90, 92, 163
 - warning 96
- Stack pane 160
 - SpeedMenu 161
- stack segments 54
- stack size 90, 92
 - committed 90
 - reserved 92
- statements
 - optimizing 101, 105
 - potential C++ errors 100
- Status Bar 11, 14
- stdarg.h 228
- stddef.h 229
- stdio FILE structures 230
- stdio.h 229
- stdiostr.h 230
- stdlib.h 230
- stepping 133, 163
 - step into 133
 - step over 134
- stop after ... warnings 101
- stop after... errors 101
- stream classes 224, 230, 257
- stream input 262
 - simple file 263
 - user-defined types 263
- stream output 259
 - filling and padding 261
 - fundamental types 260
 - I/O formatting 260
 - manipulators 260
 - simple file 263
 - user-defined types 263
- streambuf class 257
- streams 229
- string manipulation functions 231
- string move instructions 106
- string stream processing 264
- string.h 231
- strings 57, 73
- structured exceptions 204
- Style Sheets 39, 45
 - attaching 47
 - dialog box 46
 - inheriting 47
 - overriding options 49
 - setting options 45
 - sharing 47
 - between projects 48
- subexpressions 102, 104, 105

- subsystem version 94
- switch statements 105
- symbols
 - case-sensitive in library 192
 - duplication warning 96
 - in library 85
 - map files 95
 - public 93
 - stack warning 96
 - symbol declaration window 127
 - symbol tables 130
 - symbolic addresses 95
 - symbolic constants 72
 - viewing 126, 127
 - visible 128
- syntax
 - MAKE 173, 178, 179, 180, 181, 182
- syntax errors 52, 129
- syntax highlighting 24, 142
 - color options 24
- sys\locking.h 231
- sys\types.h 231

T

- TargetExpert 52
 - options 35, 38
- targets
 - adding 38
 - deleting 38
 - Make Node command 51
 - multiple 177
 - multi-target projects 37
- templates
 - instance generation 71
 - options 71
- Test stack overflow option 80
- third-party libraries 99
- this pointer 74
- threshold 54, 55
- throwing exceptions 198
- time.h 232
- tiny memory models 54, 56
- Tool Options dialog box 41
- tools 41, 173
 - adding 41, 42
 - customizing 42
 - sharing between projects 47
 - TOUCH 174
- TOUCH 174
 - command-line options 175
- trailing segments 87
- translators 41

- adding 41
- type information
 - errors 277
- typecasting
 - explicit casts 70

U

- !undef 188
- u compiler option 77
- underscores (_) 77
- uninitialized data 60
- uninitialized trailing segments 87
- UNIX compatible constants 232
- UNIX System V 78, 229
- unreachable code 100
- unsigned character types 66, 75
- user-defined warnings 99
- Using PLIB response files 193
- Using PLINK 167
 - with PCC.EXE 169
- utilities 173
 - TOUCH 174

V

- V compile option 71
- v compiler option 79
- V0 compiler option 71
- V1 compiler option 71
- Va compiler option 67
- values.h 232
- varargs.h 233
- variable live range analysis 105
- variables 156
 - compiling options 54, 58, 74, 77
 - examining 125, 155
 - optimizing 102, 104, 105
 - scope 66
- Vb compiler option 67
- VC compile option 66
- Vc compiler option 66
- Vd compiler option 66
- Ve compiler option 70
- verbose, linker option 92
- Vf compiler option 55
- Vh compiler option 67
- vi compiler option 80
- viewers, adding 41
- viewing
 - breakpoints 140
 - errors 52
 - project options 50
- virtual base pointers 65, 67

- virtual tables 55
 - corrupted 87
 - far 55, 62
 - linkage 71
 - options 71
 - pointers 55, 68
 - segments 62
- visible symbols 128
- Vmd compiler option 70
- Vmm compiler option 70
- Vmp compiler option 70
- Vms compiler option 70
- Vmv compiler option 70
- Vp compiler option 67
- Vs compiler option 71
- Vt compiler option 68
- Vv compiler option 65

W

- w compiler option 98
- warnings 267
 - alphabetical listings 271, 276
 - compiler 98
 - disabling 98
 - displaying 98
 - general 99
 - inefficient C++ coding 99
 - inefficient coding 100
 - linker 96
 - obsolete C++ 100
 - portability 100
 - potential 101
 - potential C++ 100
 - project options 98
 - stop after... 101
 - user-defined 99
- Warnings linker options 96
 - duplicate symbol 96
 - no stack 96
- watch 148, 149
 - address error 274
 - changing properties 151
 - deleting 152
 - disabling and enabling 152
 - length error 274
- Watches window 148
- Windows platforms 62
- Windows version 94
- wmsg compiler option 99
- word alignment 59

X

-X compile option 76
-x compiler option 68
-xc compiler option 68
-xd compiler option 68
-xf compiler option 68
-xp compiler option 68

Y

-y compiler option 79

Z

-Z compiler option 106
-zA compiler option 59

-zB compiler option 60
-zC compiler option 59
-zD compiler option 60
-zE compiler option 61
-zF compiler option 61
-zG compiler option 60
-zH compiler option 61
-zP compiler option 59
-zR compiler option 60
-zS compiler option 60
-zT compiler option 60
-zV compiler option 62
-zW compiler option 62
-zX compiler option 62
-zY compiler option 62
-zZ compiler option 62