CenterLine-C++
Programmer's Guide and Reference

Version 2

CenterLine Software, Inc.
10 Fawcett Street
Cambridge, Massachusetts 02138

This book was written, illustrated, and produced using FrameMaker® workstation publishing software.

This book was written, illustrated, and produced using FrameMaker® workstation publishing software.

Some parts of this manual are based on information in *AT&T C++ Language System* documents.

Distribution

The CenterLine GNU Debugger and the CenterLine C Preprocessor are free; this means that everyone is free to use them and free to redistribute them on a free basis. They are not in the public domain; they are copyrighted and there are restrictions on their distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of the CenterLine GNU Debugger or CenterLine C Preprocessor that they might get from you. The precise conditions are found in the GNU General Public License that appears in Appendix A.

If you have access to the Internet, you can get the latest distribution version of the CenterLine GNU Debugger or the CenterLine C Preprocessor via anonymous login from the following host:

ftp.centerline.com

The following file on that host contains the source for the CenterLine GNU Debugger:

/pub/TOOLS/PDM.TAR.Z

The following file on that host contains the documentation for the GNU Debugger:

/pub/Doc/gdb-info

The following file on that host contains the source for the CenterLine C Preprocessor:

/pub/TOOLS/CLPP.TAR.Z

If you do not have access to the Internet, send mail to CenterLine, and we will send you instructions on how to obtain a copy. The address is as follows:

CenterLine Software, Inc.
10 Fawcett Street
Cambridge, Massachusetts 02138

About this manual

What this manual is about

This manual is a guide to using the CenterLine-C++™ compilation system and **pdm**, our process debugger. It also contains reference information for the debugger commands.

What you should know before starting

We designed this book for readers who are familiar with the C++ programming language, an operating system like UNIX®, and a graphical user interface based on either Motif™ or OPEN LOOK®.

This manual does not describe the C++ language in detail. For a complete description please refer to the *AT&T C++ Language System Product Reference Manual*, which we provide with CenterLine-C++.

We've listed several other books about C++ and object-oriented programming in "Suggested reading"on page vi.

For more information

The C++ language supported by CenterLine-C++ at the time of release of this manual is Release 3.0.2 of the AT&T C++ Language System. Check the *Release Bulletin* or *Platform Guide* accompanying your software for the version supported by your software. The CenterLine-C++ documentation includes two manuals shipped by AT&T:

- The *AT&T C++ Language System Product Reference Manual* provides a complete definition of the C++ language supported by Release 3.0 of the C++ Language System.
- The *AT&T C++ Language System Library Manual* describes the class libraries shipped with Release 3.0.

We also provide some sections of the *AT&T C++ Language System Release Notes* and *AT&T C++ Language System Selected Readings* online, together with README files provided by AT&T. You can view these files using the Man Browser, which is described in "Using the Man Browser"on page 122. You can also find them in this directory:

full_path/**CenterLine/clc++/docs**

where *full_path* represents the path to your CenterLine directory.

The *CenterLine-C Programmer's Guide and Reference* describes the CenterLine-C compiler.

Installing and Managing CenterLine Products describes how to install CenterLine-C++ and administer it, including how to reserve licenses for particular users.

See your *Platform Guide* for system requirements and other information specific to your platform.

See the *Release Bulletin* for information generated too late to be included in the other manuals.

Suggested reading

You may find these books discussing C++ and object-oriented programming useful:

The Annotated C++ Reference Manual, Ellis & Stroustrup, Addison-Wesley 1991, 0-201-51459-1.

The C++ Programming Language (2nd. edition), Stroustrup, Addison-Wesley 1991, 0-201-53992-6.

A C++ Primer (2nd Edition), Lippman, Addison-Wesley 1991, 0-201-54848-8.

C++ Programming Guidelines, Plum & Saks, Plum / Hall Publishers 1991, 0-911-537-10-4.

C++ Programming Style, Cargill, Addison-Wesley 1992, 0-201-56365-7.

C++ Strategies and Tactics, Murray, Addison-Wesley 1992, 0-201-56382-7.

Effective C++, Meyers, Addison-Wesley 1992, 0-201-56364-9.

The C++ Answer Book, Hansen, Addison-Wesley 1989, 0-201-11497-6.

A C++ Toolkit, Shapiro, Prentice Hall 1991, 0-13-127663-8.

Advanced C++ Programming Styles and Idioms, James Coplien, Addison-Wesley 1992, 0-201-54855-0.

An Introduction to Object-Oriented Programming, Budd, Addison-Wesley 1991, 0-201-54709-0.

Object-Oriented Design with Applications, Booch, Benjamin Cummings 1991, 0-8053-0091-0.

Object-Oriented Programming Using C++, Pohl, Addison-Wesley 1993, 0-8053-5382-8.

Object Orientation: Concepts, Languages, Databases, User Interfaces, Khoshafian & Abnous, John Wiley 1990, 0-471-51801-8.

Documentation conventions

Unless otherwise noted in the text, we use the following symbolic conventions and terminology:

literal names	Bold words or characters in command descriptions represent words or values that you must use literally. Bold words in text also indicate the first use of a new term.
user-supplied values	Italic words or characters in command descriptions represent values that you must supply. Italic words also indicate emphasis.
`sample user input`	In interactive examples, information that you must enter appears in **`this typeface.`**
`output/source code`	Information that the system displays appears in `this typeface.`
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
<<*none*>>	In a “Description” section, indicates how a command performs with no arguments.
Display a menu	For pull-down menus, move the mouse pointer over the menu title and press the Left mouse button (Motif GUI) or the Right mouse button (OPEN LOOK GUI). For pop-up menus, press the Right mouse button.
Select a menu item	Drag the mouse pointer to the specified menu item and then release the mouse button.
Select a button	Move the mouse pointer over the button and click the Left mouse button.
Select text	Press the Left mouse button and drag the mouse pointer over the specified text.

Examples directory

We provide a set of examples that we use throughout this manual. To install the examples, use the **c++examples** command as described in “Setting up the examples directory”on page 9.

Contents

List of Tables

List of Figures

1 Introduction to CenterLine-C++

This chapter introduces you to CenterLine-C++.
We cover the following topics:

- *Invoking CC and the debugger*
- *Version of C++ supported*
- *Our underlying C compiler*
- *Runtime libraries and header files*
- *Our process debugger*
- *CenterLine-C++ processes*
- *Setting up the examples directory*

Introduction to CenterLine-C++

CenterLine-C++ is a complete compilation system and debugger for C++ and C. It uses the CenterLine-C++ preprocessor, **clpp**, for preprocessing, the AT&T C++ Language System translator, **cfront**, for syntax and type checking, and **clcc**, the CenterLine-C compiler, for code generation.

CenterLine-C++ contains the C++ compilation system that we provide with our comprehensive C and C++ development environment, ObjectCenter. CenterLine-C++ also contains **pdm**, the symbolic debugger used in ObjectCenter's process debugging mode. The **pdm** debugger is used for debugging fully linked executables and has features similar to debuggers like **gdb** and **dbx**. It has a set of graphical browsers to make debugging and rebuilding your programs easier.

Invoking CC and the debugger

To invoke the compiler, use the **CC** command on the command line or in a makefile:

```
% CC my_program.C
```

To invoke the graphical user interface to the debugger, use the **centerline-c++** command:

```
% centerline-c++
```

Both of these commands can be invoked with various switches and arguments. For more information about **CC**, see Chapter 2, "Compiling with CenterLine-C++," on page 11 and for more information about **centerline-c++**, see Chapter 5, "Introduction to the Debugger: A Tutorial" on page 99.

Advantages of using CenterLine-C++

Using CenterLine-C++ instead of another C++ compilation system offers the following advantages:

- CenterLine-C++ lets you avoid unnecessary recompilation of common header files, which can save significant compilation time. See “Precompiled header files” on page 21 for more information about using this feature.
- CenterLine-C++ allows you to generate only the code that is actually used. Using demand-driven code generation produces smaller object modules. Smaller modules take up less disk space, link faster, and load into a debugger faster. See “Demand-driven code generation” on page 27 for more information.
- Object files compiled using CenterLine-C++ contain more debugging information, allowing improved debugging of object code.
- CenterLine-C++'s translator places more reliable line number information in object files.
- CenterLine-C++ places header-file dependency information in object files. This may not happen with other C++ translators.

  This means that if a header file is changed and you issue **make**, affected object files will be automatically recompiled if they were initially compiled using CenterLine-C++'s translator. If you had used another C++ translator, they might not be recompiled.

Version of C++ supported

CenterLine-C++'s translator is compatible with the C++ translator as defined by Release 3.0.2 of the AT&T C++ Language System. The *AT&T C++ Language System Product Reference Manual*, which is supplied with CenterLine-C++, provides a full description of the C++ language.

New support for templates

The major new feature introduced since Release 2.1 is the implementation of **template classes and functions**. Bjarne Stroustrup originally presented the template design in *Parameterized Types for C++* at the USENIX C++ Conference in Denver in October 1988. The current implementation conforms to the draft submitted to and preliminarily accepted by the ANSI C++ standards committee. For more information about templates, see Chapter 4, “Using Templates” on page 55.

Additional new features

Other new or enhanced features introduced in this release include the following:

- This release completes the implementation of true nested scopes introduced in Release 2.1.
- Constructors that can be called with no arguments by virtue of having default arguments can now be considered default constructors.
- Overloaded prefix and postfix increment and decrement operators are correctly handled.
- The extension of the dominance rule to data and enumerators as well as functions is implemented.
- The use of constructor syntax for built-in types and protected derivations is implemented.
- The following implementation details have been reworked: the front end symbol table, type checking, function matching, operator overloading, and user-defined conversions.

This release is source- and link-compatible with Release 2.0 and Release 2.1 of the AT&T C++ Language System.

For more information about compatibility with previous releases of the translator and future compatibility, see the *AT&T C++ Language System Release 3.0.2 Release Notes*. Relevant sections of this document are available online using the Man Browser, which is described in “Using the Man Browser” on page 122. You can also access them directly in the **CenterLine/clc++/docs** directory.

Our underlying C compiler

The CenterLine-C compiler (invoked with **clcc**) is an ANSI C optimizing compiler designed to achieve small code size, high-speed code execution, and fast compilation. The compiler is also compliant with K&R C (Kernighan & Richie C, also called Classic C) and is link-compatible with Sun / SPARC and HP compilers. We supply a C library and header files that conform to the ANSI C standard. For more information about the CenterLine-C compiler, see the *CenterLine-C Programmer’s Guide and Reference*, which is supplied with CenterLine-C++.

You can choose to use your own C compiler by setting the value of the environment variable **ccC**, described in “Environment variables used by CC” on page 34.

Run-time libraries and header files

We provide static, shared, and profiling versions of the C++ run-time library, **libC**, that is supplied with the AT&T C++ Language System. We also provide the complex mathematics library, **libcomplex**. We do *not* supply the AT&T task library.

We discuss the location of these libraries and how to link to them in "Using libraries and header files" on page 29. The libraries are described in detail in the *AT&T C++ Language System Library Manual*, which we provide with CenterLine-C++.

The CenterLine-C ANSI C library is installed in **CenterLine/clcc/***arch-os***/lib/libc.a**, and is described in the *CenterLine-C Programmer's Guide and Reference*.

Our process debugger

CenterLine-C++'s process debugger, **pdm**, enables you to examine a program while it executes. You can use the debugger to debug an executable file, a corefile, or a running process. When you invoke the debugger, you can choose a Motif or OPEN LOOK graphical user interface. You can also use a command-line (ASCII) interface.

Here are some of the tasks you can perform using **pdm**:

- Edit source code using integrated **vi** and GNU Emacs text editors
- Set breakpoints, conditional breakpoints, and action points
- Enter debugging commands in a Workspace that also supports evaluation of simple expressions and **tcsh** and Emacs features
- Find and fix compiler and **make** errors using the Error Browser
- Understand unfamiliar code by looking at the graphical representation of its data structures in the Data Browser, which also provides information updates during execution
- Add custom buttons and commands to the user interface

For a sample debugger session, see Chapter 5, "Introduction to the Debugger: A Tutorial," on page 99 and for how to perform debugging tasks, see Chapter 6, "Debugging with CenterLine-C++" on page 117.

2.0.2

CenterLine-C++ processes

CenterLine-C++ consists of several separate processes, as shown in Figure 1:

- The CenterLine Message Server (CLMS) is a multicast message delivery service for exchanging data among the other CenterLine processes.
- The graphical user interface (GUI) is a set of browser windows for debugging your program. The browser windows include the Main Window, Data Browser, Error Browser, and Man Browser. You can select either a Motif or OPEN LOOK GUI.
- The CenterLine Engine is the debugger itself (**pdm**), which operates on externally linked executables.
- The Edit Server translates edit requests and responses between the debugger and your editor. CenterLine-C++ includes edit servers for both **vi** and **emacs**.
- The compiling system processes compile C++ code. For an overview of the phases of the CenterLine-C++ compiling system, see page 14.

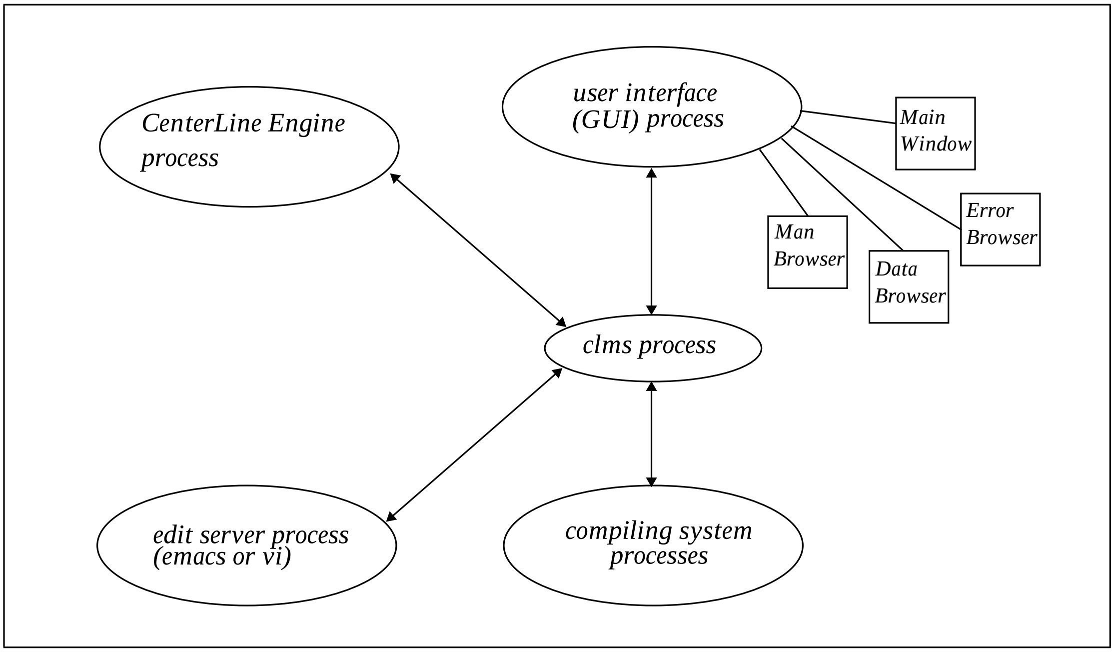


Figure 1 CenterLine-C++ Processes

For information on managing CenterLine-C++ processes, refer to *Installing and Managing CenterLine Products* and the **clms**, **clms_query**, and **clms_registry** entries in the Man Browser.

Setting up the examples directory

We provide a set of examples that we use throughout this manual. To create the examples directory in your home directory, invoke the **c++examples** command as follows:

```
% cd
% c++examples
```

If the operating system does not find the **c++examples** command, then the **CenterLine/bin** directory is not in your path. Ask your system administrator where the **CenterLine/bin** directory is on your system.

The **c++examples** command creates a directory called **c++examples_dir** in the current directory and copies the examples files to the new directory. We supply a Makefile that you can use to make the examples that we use in Chapter 5, “Introduction to the Debugger: A Tutorial” on page 99.

2 Compiling with CenterLine-C++

This chapter describes how to invoke the CenterLine-C++ compilation system and lists the command-line switches that you can use with CC. It also covers the following topics:

- *Phases of the C++ compilation system*
- *Using gprof to generate profiling information*
- *Precompiled header files*
- *Demand-driven code generation*
- *Using libraries and header files*
- *Environment variables used by CC*

Invoking CC

The **CC** command invokes the CenterLine-C++ compilation system. This is the syntax of the **CC** command line:

CC [*switches* **]** *filename* **...**

The installation process installs the **CC** command in the directory **CenterLine/bin**, which could be anywhere on your system. See your system administrator if you don't know where **CenterLine/bin** is on your system.

For example, this command line compiles **my_prog.C** with debugging information (with the **-g** switch) and produces an executable with the default name **a.out**. The **-I** and **-L** switches direct the preprocessor to search for header files in the directory **/usr/include/X11R5** and the linker to link in the library in **/usr/lib/X11R5**.

```
% CC -g -I/usr/include/X11R5 -L/usr/lib/X11R5 my_prog.C
```

Getting information about CC

Table 1 on page 16 lists the switches you can use with **CC**. You can also find this information by typing the following command at the shell prompt:

```
% man CC
```

The CenterLine installation process installs manual pages in the **CenterLine/man** directory. If the **man** command does not find the CenterLine manual page for **CC**, **CenterLine/man** may not be in the **man** command's search path. Ask your system administrator, or, if your UNIX system supports the MANPATH environment variable, add the **CenterLine/man** directory to the variable. For example:

```
% setenv MANPATH ${MANPATH}:dir/CenterLine/man
```

where *dir* is the path to your CenterLine directory.

File suffixes

The **CC** command accepts input files ending in **.c**, **.C**, **.cpp**, **.cxx**, **.cc**, or **.i**. It assumes the **.i** files are the output of the preprocessor. **CC** also accepts **.s** and **.o** files and passes them on to the C compiler.

Phases of the CenterLine-C++ compilation system

The **CC** command invokes a command-line parser and a driver. The driver invokes the other components of the CenterLine-C++ compilation system:

- The CenterLine ANSI C preprocessor, **clpp**, produces a preprocessed version of your program in a temporary file with the suffix **.i**. The preprocessor is described in more detail in Chapter 3, "Preprocessing."

NOTE You can use ANSI C preprocessing features such as token pasting and string literal expansion whether or not you choose to generate ANSI C code.

- The translator, **cfront**, performs syntax and type checking on the **.i** files produced by the preprocessor and produces temporary C versions of the files with the suffix **..c**. (On some platforms, the files have the suffix **.i**.) **cfront** also creates a temporary map file containing data type information, and produces additional symbol table information for debugging purposes if you used the **-g** switch.
- If your code uses C++ templates, the compile-time template processor, **ptcomp**, merges the map file created by **cfront** into the template repository. For more information about templates, refer to Chapter 4, "Using Templates."
- The CenterLine-C compiler, **clcc**, generates assembly code in a temporary assembly source file with the suffix **.s**.
- The assembler provided with your platform, **as**, compiles the C assembly code into object code with the suffix **.o**.
- The link-time template processor, **ptlink**, retrieves information from the template repository and may create additional object files in the repository if templates need to be instantiated.
- The linker provided with your platform, **ld**, produces an executable, called **a.out** by default, that includes start-up routines and C and C++ library routines. (Startup routines are in **/lib/crt0.o** on most platforms.)

- Depending on your platform, **patch** or **munch** links constructors and destructors of nonlocal static objects in the executable or shared library.
- Diagnostic messages are filtered through **c++filt**, which decodes ("demangles") tokens which look like C++ encoded symbols.

By default, **CC** invokes the CenterLine-C++ preprocessor, **clpp**, and the CenterLine-C compiler, **clcc**, if it is supported on your platform. You can use a different C compiler by setting the value of the environment variable ccC, as described in "Environment variables used by CC" on page 34.

You can use a different preprocessor by setting the value of the environment variable cppC. You can also override the value of the cppC environment variable with the **-Yp** command-line switch.

Examining your code at each phase of compilation

CC provides several command-line switches that let you view the output of various stages of the compilation system.

- The **-P** switch runs only the preprocessor on the code and saves a copy of the output *without* #**line** directives in a file with the **.i** suffix.
- Alternatively, the **-E** switch, used with the **-.***suffix* switch, runs only the preprocessor and saves a copy of the preprocessed file *with* **#line** directives in a file with the suffix you specify. If you don't use **-.***suffix*, the result of preprocessing is sent to standard output.
- **CC** places a temporary copy of the C code generated by the preprocessor and translator in a file with the suffix **.c** in the **/usr/tmp** directory. The **+i** switch saves a copy of this file (without #**line** directives) in the current directory with the name *file***..c** (note there are two dots before the c suffix). The **+i** switch *does not* interrupt processing.
- Alternatively, the **-F** switch, used with the **-.***suffix* switch, runs only the preprocessor and translator on your code and saves the ouput in a file in the current directory with the name *file.suffix*. If you don't use **-.***suffix*, the result of preprocessing is sent to standard output.
- The **-S** switch (a C compiler switch) saves a copy of the assembly source file in a file with the **.s** suffix, but does not assemble the code.

By default, **CC** places temporary files generated in the course of compilation in the **/usr/tmp** directory. You can override this default by changing the value of the TMPDIR environment variable. Setting environment variables is described on page 34.

CC command-line switches

Table 1 describes the switches to the **CC** command.

NOTE In addition to the switches in Table 1, **CC** accepts other switches and passes them on to the C compilation system tools. See the **clpp** manual page or Table 5 on page 52 for preprocessor switches, the **clcc** manual page for C compiler switches, and the **ld** manual page for link editor switches.

Table 1 CC Command-Line Switches

Name of Switch	What The Switch Tells CC to Do
-C	Do not discard comments; pass them through to the output file.
-dd=[on | off]	Use demand-driven code generation exclusively (**-dd=on**); this is the default setting. See the “Demand-driven code generation” section on page 27 for more information.
-dryrun	Show but do not execute the commands constructed by the compilation driver.
-ec *string*	Pass *string* to the C compiler. Be sure to use double-quotes if necessary to pass spaces or other characters significant to the shell. For example, **-ec -fsingle** passes **-fsingle** to the C compiler.
-el *string*	Pass *string* to the linker. Be sure to use double-quotes if necessary to pass spaces or other characters significant to the shell. For example, **-el** "**-a archive**" passes **-a archive** to the linker.
-E	Run only the preprocessor on the C++ source files and send the result to standard output.

Table 1 CC Command-Line Switches (Continued)

Name of Switch	What The Switch Tells CC to Do
-F	Run only the preprocessor and **cfront** on the C++ source files, and send the result to standard output. The output contains **#line** directives.
-flags_cc=*string*	Pass *string* to the C compiler. The **-ec** switch now provides similar functionality. The **-flags_cc=***string* switch is provided for backwards compatibility with previous versions of the CenterLine-C++ compiler.
-flags_cpp=*string*	Pass *string* to the C preprocessor. For instance, if you want to use the pre-ANSI rather than the ANSI C preprocessor, use the following form of this switch: **-flags_cpp=-traditional**. Be sure to use double-quotes if necessary to pass spaces or other characters significant to the shell.
-g	Produce additional symbol table information for debugging purposes.
-gdem	Demangle struct member and local variable names except where ambiguous.
-hdrepos=*directory*	Use *directory* as a repository for precompiled header files, and look in *directory* for the *filename* (precompiled header information file) used with **+k[=***filename*]. See the “Precompiled header files” section on page 21 for more information.
-ispace	Causes less inlining by decreasing inline cutoff. This in general decreases program speed but makes the program smaller. Inlining of very small inline functions continues to be done.
-ispeed	Causes more inlining by increasing inline cutoff. This in general increases program speed at the expense of increased space.
-ncksysincl	Do not check timestamps of files included with angle brackets (**< >**) when determining if a precompiled header file is out of date. See the **+k** switch (below) and also the “Precompiled header files” section on page 21 for more information.
-nCenterLine	Generate code without CenterLine extensions, including demand-driven code generation, CenterLine built-in functions, and CenterLine debugging information.

Table 1 CC Command-Line Switches (Continued)

Name of Switch	What The Switch Tells CC to Do
-pg	Enable profiling. When you use the **-pg** switch, **CC** sets the value of the LIB_ID environment variable to **C_p** so that your code explicitly links to the profiling version of the C++ library, which is named **libC_p.a**. See the "Using gprof to generate profiling information" section on page 20 for more information. NOTE: **CC** does not change the value of LIB_ID if it has been explicitly set by the user.
-pta, -ptd*pathname* **-ptf, -pth, -pti** **-ptk, -ptm***pathname* **-ptn, -pto***pathname* **-ptr***pathname***, -ptt** **-pts, -ptv**	These switches affect the template instantiation process. See Table 6 on page 78 for more information about these particular switches, and see Chapter 4, "Using Templates" for more information about templates generally.
-set_lib_id=*value*	Set the value of the LIB_ID environment variable to value. See Table 2 on page 34 for more information about LIB_ID.
-.*suffix*	When used in combination with **-E** or **-F**, place the output from each input file in a file with the specified suffix in the current directory.
-v	Verbose mode. Print the command line for each process as it begins to execute.
-Yp,*pathname*	Use *pathname* as the location of the C preprocessor. This switch overrides the value of the cppC environment variable. The default value of cppC is **$CCROOTDIR/clpp**.
+a[0 | 1]	The C++ compiler can generate either ANSI C or K&R C declarations. The **+a** switch specifies which style of declarations to produce. The default, **+a0**, causes the compiler to produce K&R C-style declarations. The **+a1** switch causes the compiler to produce ANSI C-conforming declarations. Note that this switch affects only the compiler. The **clpp** ANSI C preprocessor provides ANSI preprocessing features whether or not you use the **+a** switch.
+d	Do not inline-expand functions declared inline.
+e[0 | 1]	Only to be used on classes for which virtual functions are present, and all the virtual functions are either inline or pure. In this circumstance, this switch optimizes a program to use less space by ensuring that only one virtual table per class is generated. Specifically, **+e1** causes virtual tables to be external and defined. The **+e0** switch causes virtual tables to be external but only declared. **CC** ignores this switch for any class that contains an out-of-line virtual function.

Table 1 CC Command-Line Switches (Continued)

Name of Switch	What The Switch Tells CC to Do
+i	Leave the intermediate **..c** files in the current directory during the compilation process (note that there are two dots before the c suffix). These files do not contain any preprocessing directives, although the files passed to the C compiler do. When templates are used, it causes the instantiation system to leave **..c** files in the template repository (by default, **ptrepository**).
+k[=*filename***]**	Save and restore header files from a repository; if *filename* is provided, use it to determine which header files to save and restore. By default this switch is not set, meaning do not save and restore header files from the repository. See the **-ncksysincl** switch elsewhere in this table, and also see the "Precompiled header files" section on page 21 for more information.
+p	Disallow all anachronistic constructs. Ordinarily the translator warns about anachronistic constructs; under **+p** (for "pure"), the translator will not compile code containing anachronistic constructs. See the *AT&T C++ Language System Product Reference Manual* for a list of anachronisms.
+V	Cause calls to operator **new** to behave as in standard versions of **cfront** 3.0. This is the default behavior unless you compile with **-g**. Note however that if you specify **-g** (without **+V**) CC generates calls to **centerline_new** and / or **centerline_vec_new** to enable additional run-time error checking. These calls will generate errors if your code is not linked with the CenterLine C++ library. Use **+V** when you specify **-g** if you must link your code with other C++ libraries or if you plan to export library code to other users.
+w	Warn about constructs that are likely to be mistakes, be nonportable, or be inefficient. Without the **+w** switch, the compiler issues warnings only about constructs that are almost certainly errors.
+xfile	Read a file of size and alignments created by compiling and executing **szal.c**. The form of the created file is identical to the entries in **size.h**. This option is useful for cross compilations and for porting the translator.

Concatenating switches

You can concatenate some switches, but only the last switch in a concatenation can take an argument.

Position-independent switches

Some switches are “positionally independent”; that is, they apply to all files on the **CC** command line. For example, the following switches (some of which are **CC** switches, some of which are passed to the compiler, preprocessor, or linker) can be placed anywhere on the command line:

+a, **-dryrun**, **-v**, **-E**, **-F**, **-C**, **-P**, **-S**, **-c**, **-I**, **-D**, **-U**, **-Yp**, and **-g**

The following switches apply only to the files following them on the command line:

+d, **+p**, **+w**

For example, these two CC command lines are equivalent:

```
CC +d -v -g -I/my_include_dir test.C
CC +d test.C -vgI/my_include_dir
```

In the following sections, we describe the switches used to generate profiling information, to reuse precompiled header files, and to generate only code that is needed (demand-driven code generation). The switches used for template instantiation are described in more detail in Table 6 on page 78.

Using gprof to generate profiling information

CenterLine-C++ supports profiling with C++ source files, and it also provides a profiling version of the standard C++ library in **libC_p.a**. Here are the steps you must take to get profiling information on an executable file.

1 First, create the executable file with profiling enabled. To enable profiling, use the **-pg** switch with the **CC** command. Using the **-pg** switch causes the LIB_ID environment variable to be set to **C_p**, so that a profiling version of the library is linked in automatically. It also passes the appropriate switch to the linker so that it links in a static library. For example:

```
% CC -pg -c main.C
% CC -pg -o myexec main.o
```

2 Next, run the executable. When you run an executable you created with **-pg**, your program generates a profiling file, which by default is named **gmon.out**.

```
% myexec
```

3 To access the information in **gmon.out,** process the **gmon.out** file with **gprof**. We recommend that you also use **c++filt** to restore the names in **gmon.out** to the ones you used in your C++ code; if you don't use **c++filt**, you'll see the mangled names generated by the C++ translator instead.

```
% gprof myexec gmon.out | c++filt > myfile.gprof
```

See the UNIX manual page for the **gprof** command for more information.

Precompiled header files

CenterLine-C++ provides a facility that keeps track of header files that have been compiled to avoid recompiling them unnecessarily on subsequent compilations of the same program, or any program with the same header files. You use the following switches to take advantage of the precompiled header file facility:

-hdrepos =*dir_name*	Use *dir_name* as a repository, and look in *dir_name* for the *filename* specified with **+k=***filename*.
+k[=*filename***]**	Save and restore compiled header files from a repository. If a *filename* is provided, use it to determine which header files to save and restore. By default, this switch is not set.
-ncksysincl	Do not check timestamps of files included with angle brackets (< >) when determining if a precompiled header file is out of date.

NOTE The switches that control the precompiled header file mechanism are effective only with CenterLine-C++'s native C preprocessor, **clpp**. This means, for instance, that these switches will not work correctly if you change to another preprocessor by setting the cppC environment variable or using the **-Yp** command-line switch.

Using +k to reduce compilation time

You can use the **+k** switch with CenterLine-C++ to decrease compilation time for large programs with multiple header files where the header files have not changed between compiles. Using **+k** tells the C++ compiling system to use its **save-and-restore mechanism** for compiling header files. This mechanism saves and reuses an image of previously compiled code for header files used by your program.

When you use the **+k** switch with CenterLine-C++**,** the compiling system saves the state resulting from the initial compilations of ordered lists of header files in a repository (by default, **./hdrepository**) and restores that state on subsequent compilations of the same program or any program with the same ordered list of header files. This save-and-restore mechanism means that the first compilation of a program takes longer than it would otherwise, but subsequent compiles take significantly shorter time.

Specifying an information file

You can specify a **precompiled header information file** that contains information needed to restore the image of the compiled files. Each line in the information file should contain a list of filenames followed by an optional specification for a repository. Here's the format:

 # this is a comment line
filename1 filename2 ... filenameN **[-hdrepos** *repository_path***]**

where *filename1, filename2, ..., and filenameN* are header files enclosed in either angle brackets (< >) or double quotation marks (" "). Use the **#** sign to indicate a line with a comment.

For an example of an information file, see page 23.

CenterLine-C++ looks in the information file for the longest list of leading header files that matches the list at the beginning of each source file. Whenever CenterLine-C++ finds a match, it restores the files on the list from the repository instead of recompiling them.

The mechanism for saving and restoring header files requires that the **#include** directives specifying header files to be precompiled are the first items in the source file. This list of **#include** directives for the files may be preceded by and interspersed with semantically meaningless items such as comments, whitespace, and **#line** directives.

Using +k without an information file

If you do not specify a precompiled header information file, CenterLine-C++ interprets the initial text of each source file as a list of header files; as soon as CenterLine-C++ discovers text in the source file that is not whitespace, a comment, or a **#include** directive, it ends its list of header files for that source file.

NOTE You can take optimal advantage of CenterLine-C++'s precompiled header file mechanism by making sure that all the source files in your project contain an initial list of header files that match exactly in their order of inclusion.

Alternatively, you can set up one "mega-include" file that contains only the list of **#include** directives for the necessary header files; then make sure that all project files **#include** that one "mega-include" file.

Supplying the information file

Suppose you specify a precompiled header information file as follows:

```
<stdio.h> <string.h> "my_hdr1.h" -hdrepos /proj/my_repos
<stdio.h> <string.h> "my_hdr2.h" -hdrepos /proj/my_repos
<stdio.h>
```

Furthermore, say the beginning of your source file is as follows:

```
#include <stdio.h>
#include <string.h>
#include "my_hdr1.h"
#include "my_hdr3.h"
```

In this example, the compiling system saves the initial compilation results for **stdio.h**, **string.h**, and **my_hdr1.h** in the **/proj/my_repos** repository. When another compile is needed, the compiling system restores these compilation results from the repository and recompiles only **my_hdr3.h**.

Suppose you used the same precompiled header information file as in the preceding example but, instead of the preceding source file, you had a source file that begins as follows:

```
#include <stdio.h>
#include <string.h>
#include "my_hdr3.h"
```

In this example, the compiling system saves and restores the initial compilation results for **stdio.h** only. This is because there is no match in the precompiled header information file for any sequence of files except a sequence containing only the first one, **stdio.h.** CenterLine-C++ saves the initial compilation results for **stdio.h** and restores them as needed for later compilations; the **string.h** and **my_hdr3.h** header files would be recompiled during every recompilation of this source file.

Out-of-date compilations

CenterLine-C++ treats a previous compilation as out-of-date if it discovers anything that would cause the output of the C++ translator to differ, such as any of the following:

- Changes to the included files.
- If you change the arguments to any **CC** switch, such as **-D**, **-U**, **-I**, or **-dd={on|off}**, that affects the generated C source code, it causes the precompiled header mechanism to treat any files in the repository as outdated. As a result, CenterLine-C++ recompiles and saves the state of the newly compiled files rather than restoring an earlier state from the repository. Switches passed on to the C compiler or **ld** do not have this effect.
- Adding a comment causes the output of the translator to vary, so it causes a recompilation.
- When the time of the machine on which **CC** is executing is later than the time of the machine that the repository is written to, **CC** issues this warning:

```
Repository file filename newer than current time,
check machine times.
```

If this happens, **CC** does not restore the state of the earlier compilation. Instead it recompiles and saves the state of the new files and continues without error.

A header-file skipping example

If you have written any X Window System applications, you are probably well aware of the number and size of the header files involved. This example uses a module called **x.C** in the examples directory. If you haven't set up the examples directory yet, refer to "Setting up the examples directory" on page 9.

To begin, **cd** to the directory containing the examples and look at the header files in **x.C**:

```
% cd c++examples_dir
% head -18 x.C
```

Notice that **x.C** includes seven global header files and two local header files. In general, we recommend using global header files for header skipping rather than local header files.

To set up header-file skipping, you can create a skip information file that provides the information needed for the translator to restore the image of the compiled files. This information includes the names of the header files to be skipped and the repository in which the precompiled versions should be stored. The header files are listed in the exact order that they are included in the source file because the translator looks for the complete pattern when skipping header files.

To avoid your having to type in the contents of the **skip** information file, we have supplied it in the **c++examples_dir** directory. To view it:

```
% more skip
```

Notice that the file has a single line containing the name of the first six global header files, separated by spaces, in the exact order they appear in **x.C** (we show it here on two lines). At the end of the single line is the **-hdrepos** switch and **SR**, the name of the repository directory which will store the precompiled versions.

```
<X11/Xlib.h> <X11/Xutil.h> <X11/Xos.h> <X11/Xproto.h>
 <stdio.h> <iostream.h> -hdrepos ./SR
```

Recompiling with header-file skipping

Now that you've set up the skip information file, you can recompile. Although you can recompile **x.C** manually by using **CC** with the **+k=***filename* switch, we have supplied a special makefile target, **skipping**, for doing this. The skipping target recompiles **x.C** without header-file skipping and then with header-file skipping, and it also displays the time elapsed during each compile so you can see the speed improvement.

To recompile **x.C**, do the following:

```
% make skipping
```

You should see a series of four timestamps and three compiles. The first compile does not use header-file skipping, the second creates the repository for the precompiled header files, and the last compiles using header-file skipping.

In the following sample run, the normal compilation took 41 seconds and the compilation with header-file skipping took 25 seconds. The initial creation of the header-file repository took 95 seconds. You may get different results based on the configuration of your network and system.

```
....
...
Fri Jan 15 17:06:44
normal compile                                  } 41 seconds
...
Fri Jan 15 17:07:25
create header file skipping repository          } 95 seconds
...
Fri Jan 15 17:09:00
with header file skipping                       } 25 seconds
Fri Jan 15 17:09:25
```

With more complex programs that use large numbers of header files, the speed improvement can be more dramatic.

Restrictions

Precompiled header information files can be quite large. For instance, the file for **stdio.h** is about 300 kilobytes; others are much larger.

Uses of **__DATE__**, **__TIME__**, and **__FILE__** within a precompiled header file will not be caught and will contain the values of the initial compilation. Note that **__FILE__** can be different for the same file based on the directory where the compilation occurs.

If you specify a repository with the **-hdrepos** switch, you cannot use the precompiled header mechanism to save and restore nested header files enclosed in quotation marks rather than angle brackets.

For instance, suppose **main.C** contains the following:

```
#include "A.h"
```

and **A.h**, in turn, contains:

```
#include "B.h"
```

In this case, you cannot use the **-hdrepos** switch to compile **main.C**, although you can use the **+k** switch without **-hdrepos**.

Demand-driven code generation

Demand-driven code generation is the process of selectively generating code according to whether the code is actually used. The CenterLine-C++ translator supports demand-driven code generation with the **-dd=on** and **-dd=off** switch to the **CC** command.

For example, if you use only one class in a class library, CenterLine-C++ generates only the code for the class you used with **-dd=on**. With **-dd=off**, the compiling system generates code for all the classes in the library.

Switches for demand-driven generation

These switches turn demand-driven code generation on and off.

-dd=off Generate all code, whether or not it is used; do not use demand-driven code generation.

-dd=on (the default setting) Use demand-driven code generation exclusively. Generate only the code that is actually used in the module that is being compiled.

In the case of functions, generate code for any definitions that might be used externally, even if they are not used in the particular module being compiled.

In the case of classes, omit the class definition from the generated code if the class is not used.

Using demand-driven code generation

You can use the **-dd=on** and **-dd=off** switches on the **CC** command line:

```
% CC -dd=on -g my_source.C
```

or in makefile target rules for generating object code from C++ source files:

```
CC_SRCS = file1.C file2.C file3.C file4.C
CC_OBJS = ${CC_SRCS:.C=.o}
.SUFFIXES: .C .o
.C.o:
    CC +d -dd=off -g -c $<
```

If you're debugging your code, you can use **-dd=on** to save the space that debug information for unused declarations and inline function definitions would use. Use **-dd=off** if you want access to all the types and inline functions that you might like to use.

Advantages of demand-driven code

There are several advantages to using demand-driven code generation when you compile your C++ files.

- Demand-driven code generation decreases the number of debugging symbols that are generated by the C++ translator. This in turn reduces the size of object modules built by the CenterLine-C++ language system.
- Generating fewer debugging symbols also means that the C++ translator produces a smaller C language file, so that compilation by the C compiler is faster.
- Using demand-driven code generation reduces the amount of executable code generated for inline functions when compiling with the **+d** switch. The **+d** switch to the **CC** command specifies that inline functions not be expanded inline. If you compile with **+d** and **-dd=on**, **CC** will insert, as static functions, only those inline function definitions needed by the module.

Using libraries and header files

In this section we discuss the libraries and header files that we provide with CenterLine-C++ and how you access these libraries and the system libraries provided with your operating system. CenterLine-C++ automatically links in the standard libraries.

Search paths for libraries

CenterLine-C++ uses the same rules to search for libraries as your system's **ld** command. Here's a summary of the order in which directories are searched:

- If you specify a directory on the command line with the **-L***dir* switch to **ld,** the directory specified by *dir* is searched before the default directories.

  If you specify a library with the **-l** command-line switch, you can specify the directory to search for the library with the **-L** switch. The **-L** switch must precede the **-l** switch on the command line. For example,

  ```
  % CC -L/usr/lib/X11R5 -lX11 x.C
  ```

- You can use an environment variable to specify a colon-separated list of directories to search for libraries. These directories are searched *after* any directories specified on the command line with **-L**. The name of this environment variable is platform-specific; for example it is called LD_LIBRARY_PATH on Sun systems, and L_PATH on HP systems.
- Finally, CenterLine-C++ searches in the standard directories, **/lib**, **/usr/lib**, and, on some platforms, **/usr/local/lib**.

You can also use an environment variable (LD_OPTIONS on Sun systems, LDOPTS on HP systems) to specify a default set of **ld** switches. These switches are passed to **ld** as though they were entered *first* on the command line.

Search paths for header files

You can modify the search path used to locate **#include** files with the **-I** preprocessor switch. The preprocessor first searches in the directory containing the source file (for header files enclosed in quotation marks), then in the directories named with **-I**, if any, and finally in the system include directories. See "Locating header files" on page 41 for more information.

System libraries and header files

You can access the system libraries provided with your platform by using header files that declare interfaces to those libraries. These header files are usually installed in the directories **/usr/include** or **/usr/local/include**, both of which are usually in your standard search path. System libraries usually reside in the directories **/lib** or **/usr/lib**.

To use a function from a system library that's declared in **/usr/include/***system_header.h*, put this directive in your code:

```
#include <system_header.h>
```

CenterLine-C++ will include **/usr/include/***system_header.h* unless it encounters a file of the same name earlier in its search path.

Run-time libraries

The following run-time libraries are provided with CenterLine-C++:

- **libC.a**, the standard C++ library
- **libcomplex.a**, the complex mathematics library
- **libc.a**, the CenterLine-C ANSI C library

To use the functions declared in any of these libraries in your code, you must include the corresponding header files in your code. You may also need to explicitly link in the library when you compile your code, as described in "Linking to the complex mathematics library" on page 33.

NOTE We do *not* support the AT&T C++ Language System task library.

The standard C++ library

The standard C++ library, **libC.a**, includes the C++ **iostream** library and functions that handle error reporting and stack and vector types, run-time memory management, and invocation of static constructors and destructors.

The C++ iostream package is declared in **iostream.h** and other header files as shown in "The iostream header files" on page 33. It consists of several base classes that provide input/output conversion and buffering, together with derived classes that support additional features including formatted I/O to and from files, I/O through file descriptors, and "in-core" formatting, that is, storing and fetching from arrays of bytes.

The *AT&T C++ Language System Library Manual*, which is provided with CenterLine-C++, contains examples of using the iostream package and manual pages for the iostream library. You can also view manual pages by entering the **man** command, for example

```
% man ostream
```

The other functions provided by **libC.a** are declared in **generic.h**, **new.h**, **vector.h**, and other header files. The **generic.h** header file contains a set of simple macros used to create “pseudo-templates” before templates became part of the C++ language.

The complex mathematics library

The complex mathematics library implements the data type of complex numbers as a class, **complex**. The class overloads the standard input, output, arithmetic, assignment, and comparison operators, and the standard exponential, logarithmic, power, square root, and trigonometric functions. These functions are declared in the **complex.h** header file.

The *AT&T C++ Language System Library Manual*, which is provided with CenterLine-C++, contains manual pages for the complex mathematics library and examples of its use. You can also view manual pages by entering the man command, for example

```
% man cartpol
```

The ANSI C library

The *CenterLine-C Programmer’s Guide and Reference* describes the ANSI C library that we provide with CenterLine-C. This library is used only if you compile with the CenterLine-C **-ansi** switch. If an ANSI C-compliant C library is provided with your operating system software, we do not provide the ANSI C library.

The C header files that we provide with CenterLine-C are installed in the directory **CenterLine/clcc/***arch-os***/inc**, where **CenterLine** is the directory in which CenterLine software is installed, and *arch-os* is the directory specific to your operating system. You can view manual pages for C functions by entering the **man** command, for example

```
% man acos
```

Shared libraries

CenterLine-C++ supports **shared libraries** on all systems that provide them. A shared library, also referred to as a dynamic library, is a shared object file that is used as a library. Libraries whose names have a **.a** suffix are referred to as static or archive libraries. The suffixes of shared library names are platform dependent. Examples are **.so** and **.sl**. If CenterLine-C++ finds both a static and a shared library in the same directory, it uses the shared version.

At run time, a shared object can be linked to more than one executing program; all executing programs share access to a single copy of the object. Thus, using shared libraries can represent a significant savings in storage, but it may also reduce speed of processing.

Using C++ header files

The C++ header files that we provide with CenterLine-C++ are installed in the directory **CenterLine/clc++/***arch-os***/incl**, where **CenterLine** is the directory in which CenterLine software is installed, and *arch-os* is the directory specific to your operating system and machine architecture. Many of these files are standard UNIX system header files with argument types added to the function declarations. If the system header files distributed with your operating system support C++ constructs, the CenterLine directory does not contain redundant files.

CenterLine-C++ automatically links **libC** and **libc** with every C++ program. To use a function from one of these libraries, you need only include the appropriate header files in your code. For example, if you want to use the **cout** function, use the following **#include** directive:

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

The iostream header files

C++ does not have built-in input and output statements, but the iostreams package provides functions that allow you to use any number of input and output streams. The iostreams package is the major component of the C++ library, **libC**, which is linked in automatically by **CC**.

You do not have to link to the library explicitly to use iostreams functions. However, you *must* include **iostream.h** for any file that uses C++ I/O streams. For many programs, you need *only* include **iostream.h**. The **stream.h** header file is included for backwards compatibility with earlier releases of the C++ compilation system.

These are the iostream header files:

fstream.h	Declares iostreams specialized to files.
iomanip.h	Declares predefined manipulators and macros that change the format state, for example the field width and fill character, of the streams that they are in.
iostream.h	Declares basic iostream features, including **cout**, **cin**, and **cerr**.
stdiostream.h	Declares iostreams and streambufs specialized to interact with a stdio FILE and used for C and C++ interaction.
stream.h	Includes **iostream.h**, **fstream.h**, **stdiostream.h**, and **iomanip.h**, and used for backwards compatibility with earlier versions of C++.
strstream.h	Declares iostreams and streambufs specialized to arrays.

Linking to the complex mathematics library

You must compile and link to the complex mathematics library explicitly. To use the complex mathematics library in your application, you must specify **-lcomplex** on the **CC** command line:

```
CC -lcomplex my_appl.C
```

and you must include this directive in your code:

```
#include <complex.h>
```

Profiling version of run-time library

CenterLine-C++ provides a profiling version of the C++ library, **libC_p.a**. **CC** links to a profiling version of the library automatically when you generate profiling information for your program with the **-pg** switch. See “Using gprof to generate profiling information” on page 20 for more information.

Environment variables used by CC

The **CC** script uses environment variables to locate files it needs to run and for other environmental information. See Table 2 for a list of the environment variables used by **CC.**

You can override the values of these environment variables by setting them to different locations.

For example, if you are using a different C compiler, you could issue this command (from the C-shell):

```
% setenv ccC /usr/my/cc
```

This sets **/usr/my/cc** as the C compiler, instead of the default **cc**.

If you are using the Bourne shell, you can set and export the variable like this:

```
% ccC=/usr/my/cc; export ccC
```

Table 2 Environment Variables Used by **CC**

Name of Environment Variable	Default Value	Meaning
AON	**+a0**	K&R **(+a0)** or ANSI**(+a1)** C style declarations
CLCCDIR	<set via install>	Directory containing **clcc**
CCLIBDIR	<set via install>	Directory containing C++ libraries
CCROOTDIR	<set via install>	Directory containing **cfront**, **c++filt**, **CC**, **patch**, **ptcomp**, **ptlink**, etc.
CENTERLINE_CC_VERBOSE	**1**	Displays messages to aid in setting the ccC environment variable correctly
CL_REPOS_LOCK_MAX_WAIT	**7200**	Total number of seconds to wait for a precompiled header file lock
CL_REPOS_LOCK_STALE_TIME	**1440**	Minutes since last modification time of a precompiled header file lock before it is deleted

Table 2 Environment Variables Used by **CC** (Continued)

Name of Environment Variable	Default Value	Meaning
CLcleanR	**$CCROOTDIR/skip/cleanr**	Precompiled header repository cleanup
CPLUS	**-Dc_plusplus=1**	1.2 **cpp** C++ constant for backward compatibility
CPPFLAGS	Platform-specific flags, including **-Amachine -C -lang-c++ -DCENTERLINE_CLPP=1**	Flags to the preprocessor. NOTE: -DCENTERLINE_CLPP=1 is undefined if you override the value of the cppC environment variable
DEMANGLE	**1**	1 enables C++ link-time error message demangling
FS	**0**	1 if **-fs** switch is available
I	<set via install>	Directory for C++ include files
LIBRARY	**-l$LIB_ID**	Standard C++ library name
LIB_ID	**C**	Modify **LIBRARY**; the full path will be **$CCLIBDIR/lib${LIB_ID}.a**
LINE_OPT	<unset>	Set to "**+L**" to generate source line number information using the format "**#line %d**" instead of "**# %d**"
LOPT	**-L**	**cc** switch for linker library directory
LPPEXPAND	"**-l++**"	Specifies the string the command line argument "**-l++**" expands to
NM	**nm**	Location of **nm**
NMFLAGS	<unset>	Extra switches for **nm**
PTHDR	**.H, .h, .HH, .hh, .HXX, .hxx, .hpp**	List of header file suffixes **ptlink** uses to look up template type declarations
PTSRC	**.C, .c, .CC, .cc, .CXX, .cxx, .cpp**	List of source file suffixes **ptlink** uses to look up template type definitions

Table 2 Environment Variables Used by **CC** (Continued)

Name of Environment Variable	Default Value	Meaning
PTOPTS	<unset>	Default switches to be passed to the template instantiation system
TMPDIR	**/usr/tmp**	Directory used as root of temporary file directory for C++ compilation
ccC	**$CLCCDIR/clcc -w**	The C compiler (the value of ccC defaults to the native C compiler if **clcc** is not available)
cfrontC	**$CCROOTDIR/cfront**	The C++ translator
cPLUS	**-D__cplusplus=1**	2.0 **cpp** C++ constant for ANSI C conformance
cplusfiltC	**$CCROOTDIR/c++filt**	C++ link error message filter
cppC	**$CCROOTDIR/clpp**	The C preprocessor
munchC	**$CCROOTDIR/munch**	The **munch** executable
patchC	**$CCROOTDIR/patch**	The **patch** executable
ptcompC	**$CCROOTDIR/ptcomp**	The **ptcomp** executable
ptlinkC	**$CCROOTDIR/ptlink**	The **ptlink** executable
skippp	**$CCROOTDIR/skippp**	The precompiled header preprocessor

3 Preprocessing

This chapter describes the CenterLine-C++ preprocessor. We cover the following topics:

- *Header file inclusion*
- *Macro definition and expansion*
- *Conditional compilation*
- *Line control*
- *Reporting diagnostic messages*
- *Implementation-dependent behavior*
- *Preprocessor switches*

The CenterLine-C++ Preprocessor

A preprocessor manipulates the text in your source file and produces input to the compiler. This chapter describes the preprocessor distributed with CenterLine-C++.

Getting information about clpp

The CenterLine ANSI C preprocessor, **clpp**, is based on the GNU-C Compatible Compiler Preprocessor. We have enhanced it to handle CenterLine's precompiled header file facility, described on page 21. For usage information and a listing of preprocessor switches, issue the **man** command at the shell:

```
% man clpp
```

The CenterLine installation process installs manual pages in the **/CenterLine/man** directory. If the **man** command does not find the CenterLine manual page for **clpp**, **CenterLine/man** may not be in the **man** command's search path. Ask your system administrator, or, if your UNIX system supports the MANPATH environment variable, add the **CenterLine/man** directory to the variable. For example:

```
% setenv MANPATH ${MANPATH}:dir/CenterLine/man
```

where *dir* is the path to your CenterLine directory.

What is clpp?

The **clpp** preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs.

The preprocessor always does the following:

- Replaces C and C++-style comments with single spaces
- Deletes all backslash-newline sequences
- Expands all predefined macro names

In addition, the preprocessor provides the following optional facilities:

- Header file inclusion
- Macro expansion
- Conditional compilation
- Line control

Preprocessor directives implement each of these facilities.

Preprocessor directives

Preprocessor directives always begin with the **#** sign, optionally preceded by space and tab characters, followed by an identifier called the command name. They can appear anywhere in your code and can be continued over several lines by placing a backslash (\) at the end of the line to be continued.

There is a fixed set of command names, as shown in Table 3. We discuss each of the facilities these commands are used for in the rest of the chapter.

Table 3 Preprocessor command names

Preprocessor command name	Used for:
#include, #include_next	Header file inclusion
#define, #undef	Macro definition and expansion
#if, #else, #elif, #endif **#ifdef, #ifndef**	Conditional compilation
#line	Line control
#error, #warning	Reporting diagnostic messages
#pragma	Implementation-dependent behavior

Header file inclusion

Use the **#include** directive to include the contents of other files, usually header files, before your file is compiled. When the preprocessor encounters a **#include** directive, it scans the file specified for input before continuing.

NOTE CenterLine-C++ provides a facility that keeps track of header files that have been compiled to avoid recompiling. See page 21 for more information.

Locating header files

How the preprocessor locates the file depends on which of three forms the command argument takes:

#include <*filename*> — Searches for the system header file called *filename*, first in the list of directories you specify on the command line with the **-I** switch, then in a standard list of system directories.

#include *"filename"* — Searches for your own header file called *filename*, first in the directory of the current input file, then in the same directories used for system header files.

#include *identifier* — Expands any macros contained in *identifier*, then completes the header file search as above. The search path depends on whether the resulting expansion is enclosed in angle brackets or double quotes. This is sometimes called a **computed #include**.

One use of this computed **#include** might be to include a site-specific version of a header file. The following example uses macro expansion and conditional compilation, which are described in the next two sections. This sequence of directives causes the preprocessor to include a different version of a header file called **my_args.h** at the site called **paris**:

```
#ifdef paris
#define my_args "my_args.paris.h"
#else
#define my_args "my_args.h"
#endif
#include my_args
```

Nested #include directives

The files that you include with the **#include** directive can themselves contain **#include** directives. The **clpp** preprocessor supports approximately 198 levels of nesting.

Substituting other header files with #include_next

If your program relies on a system header file that doesn't behave the way you need it to on all the platforms your program supports, you can write a local version of the header file that adds to the system header file.

You can use the **#include_next** command to ensure that the preprocessor finds first your local version of the header file, then the system version. The **#include_next** command behaves like the **#include** command, but it begins its search for the header file in the next directory on the search path after the directory that contains the current file.

For example, if you want to modify the **errno.h** system header file, use this directive in your program:

```
#include <sys/errno.h>
```

In the local version of the header file, use this directive

```
#include_next <sys/errno.h>
```

Use the **-I** switch to the **CC** command to specify the directory that contains the local version of the header file, for example **-I/usr/local/include**. In this example, the preprocessor first finds the local version of the **errno.h** header file. When it encounters the **#include_next** command, it searches for the next header file in its search path called **errno.h** and finds and includes the system header file.

Macro definition and expansion

The preprocessor expands predefined macros and macros that you define using the **#define** directive. We cover the following topics in this section:

- Defining simple macros
- Defining macros with arguments
- Specifying string literals
- Concatenating tokens
- Differences between ANSI C and K&R C
- Predefined macros

Simple macros

The simplest macro definition has this syntax:

#define *macro_name macro_body*

This form is most often used to define a constant; for example, if your program includes a header file that contains this directive:

```
#define LENGTH 600
```

the preprocessor replaces each occurrence of **LENGTH** in your program with **600**. The macro definition remains in force until the end of the translation unit, or until it is undefined with an **#undef** directive.

You can define a macro that refers to another macro. For example:

```
#define WIDTH 2*LENGTH
```

This is not equivalent to defining **WIDTH** to equal **1200**, because the preprocessor doesn't replace **WIDTH** with **2*LENGTH** until you use **WIDTH**.

In C++, you can use a **const** declaration instead of a macro, for example

```
const int LENGTH=600;
```

Using **const** has the advantage of making **LENGTH** available to a symbolic debugger. Also **const** values can have type and scope like variables.

Macros with arguments

You can define a macro that accepts arguments. The syntax is as follows:

#define *macro_name(arg1, arg2,...argn) macro_body*

The opening parenthesis must follow the macro name immediately with no white space, otherwise the preprocessor interprets the white space as the macro body. The arguments can be any valid identifiers, separated by commas and optional white space. Here's an example:

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

The parentheses around the macro body are not required, but we recommend that you use them to avoid problems that can occur due to C's operator precedence rules.

To use the macro, specify its name followed by a list of arguments in parentheses, separated by commas. The number of arguments you list must match the number in the macro definition.

In C++, you can replace a macro like this with an inline or template function, which has the advantage that the function name will be available to a symbolic debugger. For example, this inline function replaces the **min** macro defined above:

```
inline int min(int a,int b)
{
    return ((a) < (b) ? (a) : (b))
}
```

Using an inline or a template function instead of a macro also allows the C++ compiler to perform type checking on any call to the function.

Specifying string literals

You can turn a macro argument into a string literal by preceding it with a # token (sometimes called the **stringizing** operator). This example defines and uses a macro called **print_name**:

```
#include <iostream.h>
#define print_name(name) cout << "My name is" #name "\n"

main () {
print_name(Anita);
}
```

After preprocessing, **main()** looks like this:

```
main () {
cout << "My name is" "Anita" "\n"
}
```

The preprocessor later concatenates adjacent strings, so the output of the program is this:

```
My name is Anita
```

Concatenating tokens

If the ## operator appears between two tokens in the macro body, the preprocessor first replaces the tokens if they are parameters, then removes the ## token and any white space surrounding it.

For example, suppose you define this macro:

```
#define size(name,no) new ## name = no * old ## name
```

If you use the **size** macro as follows:

```
size(Length,3)
```

You get the following expansion:

```
newLength = 3 * oldLength
```

ANSI C differences in macro expansion

There are several differences between the ways ANSI C style preprocessors such as **clpp** and pre-ANSI preprocessors handle macro expansion. If you're using legacy code or pre-ANSI C header files you may encounter these differences. The *Annotated C++ Reference Manual*, in its commentary on preprocessing, describes differences in detail. (See ”Suggested reading” on page vi for publication details.)

Here's a simple example to illustrate how **clpp** and pre-ANSI preprocessors handle strings, character constants, and concatenation. Suppose you have this code:

```
#define old_string(x) "x"
#define old_char(y) 'y'
#define old_join(m,z) m/* */z

#define new_string(a) #a
#define new_join(c,d) c##d
```

```
main() {

old_string(this is my string);
old_char(this is my char);
old_join(con,catenated);

new_string(this is my string);
new_join(con,catenated);
}
```

The “old” macros produce the desired result if you use a pre-ANSI preprocessor, the “new” macros if you use **clpp** or another ANSI C preprocessor. There is no ANSI C equivalent to the “charizing” feature provided with some pre-ANSI preprocessors, which replaces the contents of character constants with the spelling of their formal arguments.

A traditional preprocessor produces this output:

```
"this is my string";
'this is my char';
concatenated;

#this is my string;
con##catenated ;
```

Here’s the result when you use **clpp**:

```
"x" ;
'y' ;
con catenated ;

"this is my string" ;
concatenated ;
```

Predefined macros

CenterLine-C++ predefines the macros listed in Table 4. These macros cannot be undefined or redefined, except as noted in the table.

The **__LINE__** and **__FILE__** macros can be set by the **#line** directive, as described in ”Line control” on page 51.

To allow conditional compilation for source files that are compiled by both the C++ translator and the C compiler, CenterLine-C++ predefines the macros **__cplusplus** and **c_plusplus**. These macros are predefined to the value **1**. The **c_plusplus** macro is included only for backward compatibility with AT&T C++ 1.2 source code. When writing new code, use the **__cplusplus** macro instead of **c_plusplus**.

Table 4 Macros Recognized by CenterLine-C++

Name of Macro	Macro Definition	Additional Information
__FILE__	Name of the file being read.	Also predefined by **cc**.
__FUNC__	Name of the function being read.	We do not recommend that you use this macro, since it is not available in other C++ or C implementations.
__LINE__	Line number of the file being read.	Also predefined by **cc**.
__DATE__	Date the file was read (*"Mmm dd yyyy"*).	Defined only if the preprocessor is in ANSI C mode.
__TIME__	Time the file was read (*"hh:mm:ss"*).	Defined only if the preprocessor is in ANSI C mode.
__STDC__	Defined as **1**.[a]	Defined only if the preprocessor is in ANSI C mode.
__cplusplus	Always defined as **1**.	Defined as 1 whether using a K&R C or ANSI C preprocessor.
c_plusplus	Always defined as **1**.	Defined as 1 whether using a K&R C or ANSI C preprocessor.

a. This macro is defined by C compilers and interpreters that conform to the ANSI standard. On some platforms, such as Solaris 2.x, it can be defined as 0, and on others it can be undefined.

Refer to the *CenterLine-C Programmer's Guide and Reference* or the **clcc** manual page for a list of the predefined macros recognized by the CenterLine-C compiler.

Conditional compilation

Conditional compilation allows your program to behave differently depending on the conditions under which it is compiled. Conditional commands are most often used in three situations:

- When portions of the code differ depending on the platform on which the code will run. For example, library routines may vary among operating systems.
- When the same source file can be compiled into two or more applications.
- When a section of the code is obsolete, but you want to retain it in the source file for future reference.

Conditional directives begin with the **#if**, **#ifdef**, or **#ifndef** commands and end with **#endif**. They can also contain **#else** and **#elif** commands.

Conditional syntax

Conditional directives that begin with the **#if** command have this syntax:

#if *exp1*
text_if_exp1_true
[**#elif** *exp2*
text_if_exp2_true]...
[**#else** [*/* not exp1 and not exp2*/*]
text_if_exp1_and_exp2_false]
#endif [*/* exp1 */*]

The text following the first expression (*exp1*, *exp2*,...) that evaluates to nonzero is preprocessed and the remaining conditional directives are ignored. If none of the expressions following the **#if** and **#elif** commands are nonzero, the text following the **#else** command, if any, is preprocessed.

The optional comments after the **#else** and **#endif** commands make it easier to read nested conditional directives.

NOTE In K&R C, **#else** and **#endif** can be followed by tokens, so that, for example, the following is a legal directive in K&R C:

```
#if KERNEL
...
#endif KERNEL /* legal in K&R C */
```

For full ANSI C compliance, only comments can follow **#else** and **#endif** statements:

```
#endif /* KERNEL */
```

Limitations on the content of expressions

The expression you use in a conditional directive must be a C expression of integer type. It can contain integer constants, character constants, arithmetic operators, identifiers, and macro calls.

NOTE The preprocessor treats all identifiers that are not macros as 0. Also, the way it interprets character constants depends on the conventions of the machine and operating system on which the code is running.

You cannot test the size of a variable or data type. The preprocessor doesn't understand "sizeof" operators, "enum"-type operators, typedef names, or type keywords. In this example, **BUFSIZE** must be a macro:

```
#if BUFSIZE == 1020
...
#elif BUFSIZE == 2040
...
#else /* BUFSIZE != 2040 & BUFSIZE != 1020/
...
#endif /* BUFSIZE == 1020 */
```

Using #ifdef and #ifndef

You can use **#ifdef** or **#ifndef** with a macro name if a section of your code is relevant only under certain conditions. You can use a predefined macro or a macro you have defined yourself.

For example, if your program has sections that differ according to whether or not it's compiled in a UNIX environment, you could use this directive:

```
#ifdef unix
.
<code to be compiled if unix is defined>
.
#else /* not unix */
.
<code to be compiled if unix is not defined>
.
#endif /* unix */
```

The commands **#if defined** and **#if !defined** are equivalent to **#ifdef** and **#ifndef**, but they enable you to combine two conditions in one line. For example:

```
#if defined (_sparc_) || defined (_hp_)
.
<code to be compiled if _sparc_ or _hp_ is defined>
.
#endif
```

The parentheses surrounding the macro name are optional.

Retaining obsolete code

If you want to refer to a section of your program that you've changed, but you no longer want it compiled, you can retain it in your source code and use a conditional directive that always evaluates to false:

```
#if 0
.
<obsolete code>
.
#endif
```

Line control

The output from the preprocessor is a combination of your input files and any files you included with **#include**. The included files and any conditional directives and macros you use cause the line numbers in the preprocessor output to be different from those in the original source file.

To enable error or warning messages to indicate at what line, and in which file, inconsistencies are detected, the preprocessor uses the **__LINE__** and **__FILE__** predefined macros. They expand, respectively, to the current input line number and the file being preprocessed. After a **#include** directive, the **__FILE__** macro contains the name of the included file, until processing resumes on the file containing the **#include** directive.

A **#line** directive changes the contents of the **__FILE__** and **__LINE__** macros. This is useful if the original source file is processed by another program, such as a parser generator, and the output from that program becomes the input to the preprocessor. The parser generator can insert **#line** directives into its output so that the output from the preprocessor can refer to the original filename and line number.

For example, the following directive sets the **__LINE__** macro to **15** and the **__FILE__** macro to **my_file**:

```
#line 15 my_file
```

Reporting diagnostic messages

The **#error** directive causes the preprocessor to report a fatal error. The text following the **#error** directive is the error message. For example, if your program requires a particular condition to be true, you could test for the condition and generate an appropriate message.

Here’s an example:

```
#if SIZE != 1000
#error "SIZE must equal 1000"
#endif
```

The **#warning** directive causes the preprocessor to print a diagnostic message, but it does not interrupt processing.

Implementation-dependent behavior

The ANSI standard provides a preprocessor directive of the form **#pragma** *token-string*. Its effect is determined by the implementation, and it is ignored if the implementation does not recognize *token-string*.

The **clpp** preprocessor ignores all **#pragma** directives and passes them on to the C compiler. Refer to the *CenterLine-C Programmer's Guide and Reference* for more information about the **#pragma** directives recognized by **clcc**.

Preprocessor switches

Table 5 shows the switches accepted by the **clpp** preprocessor. Some of these switches can be used on the **CC** command line, but others must be passed to the preprocessor from the **CC** command line with the **-flags_cpp=** switch. Here's an example:

```
% CC -flags_cpp="Wtraditional -pedantic" my_prog.C
```

This command line generates warnings if your code violates certain pre-ANSI (**-Wtraditional**) or ANSI semantics (**-pedantic**).

Table 5 clpp Preprocessor Switches

Switch	Meaning
-C	Do not discard comments; pass them through to the output file. Comments appearing in arguments of a macro call will be copied to the output before the expansion of the macro call.
-D*name*	Predefine *name* as a macro, with definition **1**.
-D*name=definition*	Predefine *name* as a macro, with definition *definition*. There are no restrictions on the contents of *definition*, but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. If you use more than one **-D** for the same name, the rightmost definition takes effect.

Table 5 **clpp** Preprocessor Switches (Continued)

Switch	Meaning
-dD	Write to standard output your **#define** commands and the result of preprocessing. Do *not* list predefined macros.
-dM	Write to standard output the **#define** directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor. Assuming you have no file **test.h**, the command **touch test.h; clpp -dM test.h** will show the values of any macros predefined on your platform.
-H	Writes the name of each header file used to standard output.
-I*directory*	Add *directory* to the end of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one **-I** switch, the directories are scanned in left-to-right order; the standard system directories come after.
-I-	Any directories specified with **-I** switches before the **-I-** switch (note the hyphen after **-I**) are searched only for the case of **#include** "*file*"; they are not searched for **#include** <*file*>. If additional directories are specified with **-I** switches after the **-I-**, these directories are searched for all **#include** directives. In addition, the **-I-** switch inhibits the use of the current directory as the first search directory for **#include** "*file*". Therefore, the current directory is searched only if it is requested explicitly with **-I**. Specifying both **-I-** and **-I** allows you to control precisely which directories are searched before the current one and which are searched after.
-imacros *file*	Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of **-imacros** *file* is to make the macros defined in *file* available for use in the main input. The preprocessor evaluates any **-D** and **-U** switches on the command line before processing **-imacros** *file.*
-include *file*	Process *file* as input, and include all the resulting output, before processing the regular input file.
-lang-c **-lang-c++**	Specify the source language. **-lang-c++** includes additional default include directories for C++ and enables the preprocessor to handle C++ comment syntax.
-M	Lists dependencies of the source file on standard output. The preprocessor writes one **make** rule containing the object file name for the source file, a colon, and the names of all the files included with **#include**. If there are many included files then the rule is split into several lines. This feature is used in automatic updating of makefiles.

Table 5 clpp Preprocessor Switches (Continued)

Switch	Meaning
-MM	Same as **-M**, except that only the files included with **#include** "*file*"are listed. System header files included with **#include** <*file*> are omitted.
-nostdinc	Do not search the standard system directories for header files. Only the directories you have specified with **-I** switches (and the current directory, if appropriate) are searched.
-P	Inhibit generation of **#**-lines (lines beginning with the # sign and containing line-number information) in the output from the preprocessor. This might be useful when running the preprocessor on something that is not C code and will be sent to a program which might be confused by the **#**-lines. When used on the **CC** command line, runs only the preprocessor and sends output (without #-lines) to a file with the suffix **.i**.
-pedantic	Issue warnings required by the ANSI C standard in certain cases such as when text other than a comment follows **#else** or **#endif**.
-pedantic-errors	Same as **-pedantic**, except that errors are produced rather than warnings.
-traditional	Use pre-ANSI C rather than ANSI C style preprocessing.
-trigraphs	Process ANSI standard trigraph sequences. These are three-character sequences, all starting with ??, that are defined by ANSI C to stand for single characters. For example, ??/ stands for \, so ??/n is a character constant for a newline, \n.
-U*name*	Do not predefine *name*. If both **-U** and **-D** are specified for one name, the name is *not* predefined.
-undef	Remove initial definitions of nonstandard macros.
-Wall	Request both **-Wtrigraphs** and **-Wcomment** (but not **-Wtraditional**).
-Wcomment **-Wcomments**	Warn whenever a comment-start sequence (**/***) appears in a comment. (Both forms have the same effect).
-Wtraditional	Warn about certain constructs that behave differently in traditional and ANSI C.
-Wtrigraphs	Warn if any trigraphs are encountered (assuming they are enabled).

4 Using Templates

This chapter provides an introduction to using templates with CenterLine-C++. It includes a description of the instantiation process and the switches used for instantiation, some usage scenarios, and troubleshooting tips.

Using templates

Templates are the mechanism in C++ for supporting **parameterized types**.

Parameterized types allow you to implement generic code for a type and then implement that type with different parameters.

For example, you can define a general container type such as **List** or **Set** as a template, and specify the type of the elements in the container as a type parameter. Thus you could define a **Set** template and specify its type parameters as **int**, **Button**, or **cookbook**. As a result, the compiler could automatically create a **Set** of **ints**, a **Set** of **Buttons**, or a **Set** of **cookbooks**.

In case you are not familiar with templates, we provide some background information about how they are defined by the C++ language and how they are implemented in CenterLine-C++. In the rest of this chapter, we discuss the following aspects of templates:

- Basic concepts and syntax
- Using templates with CenterLine-C++
- The instantiation process
- Coding conventions
- Lookup schemes
- Map files
- Switches for templates
- Usage scenarios
- Specializations
- Examples
- Common pitfalls
- Troubleshooting
- Tools
- Summary of terminology

For more language information about templates, see the *AT&T C++ Language System Product Reference Manual*. For more implementation information, see “Template Instantiation in C++ Release 3.0.2”, an excerpt from the *AT&T C++ Language System Selected Readings*, which is available online in the following file:

CenterLine/clc++/docs/cfront3.0.2.templ_inst

If you are using the GUI, you can also view it in the Man Browser by selecting the “Template Instantiation” topic in the AT&T Documentation category. For more information about the Man Browser, see “Using the Man Browser” on page 122.

Basic concepts and syntax

There are two kinds of templates in C++: class templates and function templates. A class template allows you to define a pattern for class definitions; generic container classes are good examples of class templates. A function template defines a pattern for a family of related overloaded functions; the function template lets one or more of the function parameters be a parameterized type. In the next two subsections, we describe class templates first, and then function templates.

Class templates

In C++, you use a class declaration to specify how to construct an individual object. Similarly, you use a class template to specify how to construct an individual class.

Once you have specified a class template completely, the C++ language system can use the template to generate a template class, which is just like any other class. Whenever you declare an object of the template class, the language system uses the template class to create an individual object.

Creating the class from the template is called instantiating the template. See Figure 2 for a conceptual illustration of the relationship between a class template, an individual class (template class) instantiating the template, and declaring an object of the template class.

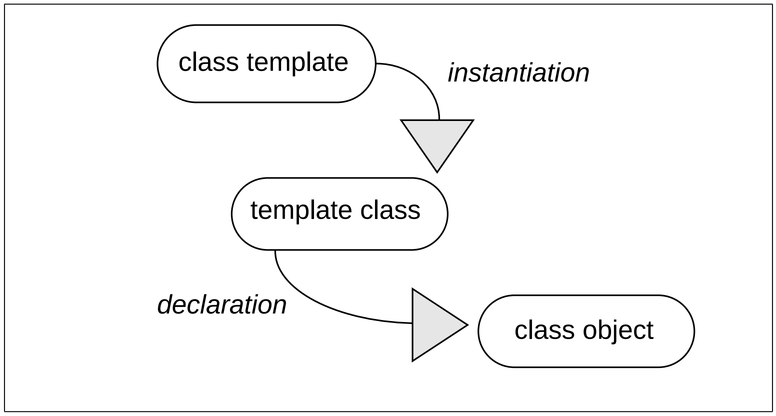


Figure 2 Instantiation and Declaration:
From Template to Class to Object

Say, for instance, you wish to create a bank that is a list of accounts. You want the bank to be an object just like any other class object in C++. You can use templates to create the bank by writing code that follows these steps:

1 Create a class template for lists in general; let's say you name it **List**, and you want it to work for all types **T**:

```
template <class T> class List
{
private:
 T *data;
 List *next;
public:
 List();        // construct a List

 List(T& type);  // constructor of a List, given a T

 List *nextLink();// return a pointer to the next
                        // item in the linked list
 void setNext(T newData); // add an item containing
                        // newData to the list
 T *thisData();         // return this List's data
}
```

Make sure that you have defined the **account** type (in another file) that you want to use as the parameterized type **T** with **List**:

```
class account
{
.
.
.
};
```

2 Declare an object of the template class from the template, using an **account** as the parameterized type, and construct an individual bank object:

```
account my_checking;
List<account> A_Bank(my_checking);
```

3 Now you can add accounts to the bank by using a **List<account>** member function:

```
account your_checking;
A_Bank.setNext(your_checking);
```

See Figure 3 for a conceptual illustration of the instantiation and declaration of a class template in the bank example.

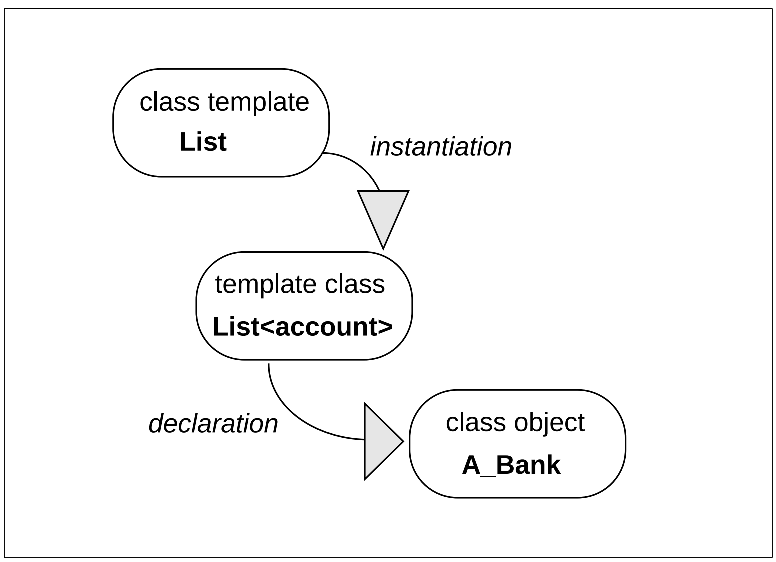


Figure 3 From **List** Class Template to **A_Bank** Class Object

Function templates

C++ allows you to overload functions — that is, you can give many functions the same name as long as each function definition is distinguished by the number and / or type of its function arguments. You can think of a function template as a shorthand way to define a set of overloaded functions.

For instance, suppose you wish to define a set of overloaded functions so that each function returns the larger of its two arguments. The simplest form of such a function is **max(int, int)**:

```
int max(int a, int b) { return (a > b) ? a : b; }
```

In addition to comparing integers, you also want to overload **max** to compare two classes of type **Circle** as well as comparing variables of the built-in types **float** and **char**:

```
Circle max(Circle a, Circle b){ return (a > b) ? a : b;}
float max(float a, float b) { return (a > b) ? a : b;}
char max(char a, char b) { return (a > b) ? a : b; }
```

Clearly each of these functions requires a definition of the greater-than operator (>); this definition is part of the language definition for **int**, **float**, and **char**, but must be defined specifically for the **Circle** class.

Here's an example of a function template that defines the same pattern as the preceding set of overloaded **max** functions:

```
template <class T> T max( T a, T b)
      { return ( a > b ) ? a : b; };
```

The data type for the **max** function template is represented by the template argument: **<class T>**. Once you define this function template, you can use the **max** function with any data type for which the operator > is defined.

See Figure 3 for a conceptual illustration of the instantiation and use of the **max** function template with **Circle** as the parameterized type. Note that the **Circle(max)** function is generated implicitly by the compiler.

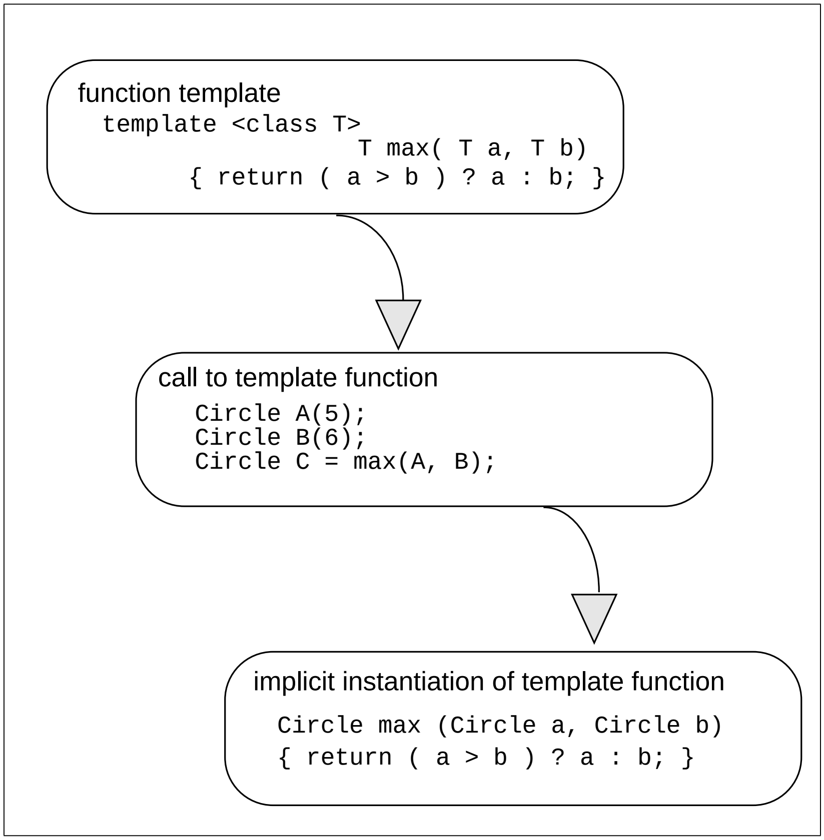


Figure 4 From Function Template to Function Call

Once you write the function template for the **max** function, using **T** as the parameterized type, your application can make a function call such as **max(A,B)**. Then, CenterLine-C++'s automatic template instantiation system implicitly creates the instantiated template function needed to implement the **max** function call. (It does not create a physical copy of the instantiated template function.)

In the case illustrated in Figure 3, the parameterized type in the function call is **Circle**, so the instantiated template function returns a **Circle** and takes **Circles** as arguments.

Using templates with CenterLine-C++

In this section we present an overview of the use of templates with CenterLine-C++; we do so by focussing on another example of a simple class template. Later in the chapter we describe the instantiation process and coding conventions in more detail. The three files used in this example, **Vector.h**, **Vector.c**, and **appVector.C**, are available online in the examples directory. See “Setting up the examples directory” on page 9 for how to install the examples directory.

Declaring a template in a .h file

Suppose that you want to use one-dimensional arrays, or vectors, that grow dynamically as new elements are added and that can contain different types as elements. You might declare a vector class template as follows:

```
template <class T> class Vector
{
   T* data;
   int size;
public:
   Vector();
   T& operator[](int);
};
```

This class template declaration has two private data members, **data** and **size**, and two public functions, **operator[]** and the constructor. There is one argument **T** to the template.

We put the template declaration of **Vector** in a file named **Vector.h**. As its suffix indicates, **Vector.h** is a header file. In the rest of this discussion, we will refer to it as the **template declaration file**.

NOTE	The template instantiation system allows you to use filenames with different suffixes. See the "Dynamic extension lookup" section on page 73 and the "Map files" section on page 75 for details on how to do so correctly.

Specifying a template implementation in a .c file

By convention, if the declaration file is named **Vector.h**, we put the implementation of the **Vector** template in a file named **Vector.c**. This file is generally referred to as the **template definition file**. You can use other suffixes for the template definition file, but the name (in this case **Vector**) must be the same as that of the template declaration file.

Here's a possible implementation for **Vector**:

```
template <class T> Vector<T>::Vector()
{
   // start off with 3 elements
   size = 3;
   data = new T[size];
}


template <class T> T& Vector<T>::operator[](int n)
{
   int os;
   int i;
   T* newdata;
   // grow if have to
   if (n >= size)
   {
         os = size;
         while (size <= n) size *= 2;
         newdata = new T[size];
         for (i = 0; i < os; i++)
                newdata[i] = data[i];
         delete [] data;
         data = newdata;
   }
   // return reference to data slot
   return data[n];
}
```

Note that the code in **Vector**'s declaration and implementation is parameterized — it uses a type that is unknown but represented by **T**. Also, note that the **.c** file does not include the **.h** file. The automatic instantiation system will locate **Vector.c** according to the rules described in "Dynamic extension lookup" on page 73.

Specifying a template class in an application

We put a simple application that uses the **Vector** template in **appVector.C**:

```
#include <iostream.h>
#include "Vector.h"

main()
{
Vector<int> v;
int i;

// put data in the vector
for (i = 1; i <= 5; i++) v[i] = i * i;

// display data in the vector
for (i = 1; i <= 5; i++)
    cout << i << "" << v[i] << \n";
}
```

This application using the **Vector** template specifies **Vector<int>**; that is, it substitutes type **int** for the **T** in the declaration and implementation of the template. **Vector<int>** is an example of a template class — a template with particular arguments — where **<int>** is a template argument.

Compiling the application

Compile this application with the **-ptv** switch so that you can see what the compile-time (**ptcomp**) and link-time (**ptlink**) template processors are doing. The instantiation process is described on page 67.

```
% CC -ptv appVector.C
CC  appVector.C:
CC[ptcomp] locked repository [1] ...
CC[ptcomp] read raw cfront information [0] ...
CC[ptcomp] read old map file [0] ...
CC[ptcomp] made list of unique filenames in new map file [0] ...
CC[ptcomp] deleted old map file entries [0] ...
CC[ptcomp] added in new map entries [0] ...
CC[ptcomp] wrote new map file [1] ...
CC[ptcomp] unlocked repository [0] ...

/* compiler output deleted */...

CC[ptlink] locked repository [0] ...
CC[ptlink] read name map file list [0] ...
CC[ptlink] read in all objects and archives [1] ...
CC[ptlink] finished link simulation to pick up undefineds [0] ...
CC[ptlink] made list of unique template class names [0] ...
```

```
==========left to do==========
                      Vector<int>
CC[ptlink] now looking at template Vector<int> [0] ...
CC[ptlink] wrote instantiation file Vector__pt__2_i.c [0] ...
CC[ptlink] rebuilt header file cache [1] ...
CC[ptlink] did dependency check on Vector__pt__2_i.c (missing object)
[0]...
CC[ptlink] CC line is:

/* compiler output deleted */...

CC[ptlink] compiled Vector__pt__2_i.c [7] ...
CC[ptlink] added symbols from Vector__pt__2_i.o to symbol table [0] ...
CC[ptlink] finished link simulation to pick up undefineds [1] ...
CC[ptlink] unlocked repository [0] ...
```

The CC command compiles **appVector.C** and goes on to create an object file for the template class with the parameters specified in **appVector.C**. The object file instantiating **Vector<int>** contains the template class members **Vector<int>::Vector()** and **Vector<int>::operator[](int)**. This process is called **instantiation**.

You may get syntax errors at this stage if CenterLine-C++ finds errors in your template definitions. Make corrections in the **.c** file that implements the template containing the error, in this case, the **Vector.c** file, rather than in your application file.

For instance, if you get an error message indicating “missing template arguments”, you may have forgotten to specify the parameter for a template in your template definition file. This could be a matter of simply remembering to write **<T>** after the template name.

After you successfully compile and link your program, you can run it. Here are the results when you run **appVector.C**:

```
% a.out
1 1
2 4
3 9
4 16
5 25
```

The instantiation process

NOTE In most cases, you can use templates without knowing the details of the instantiation process. However, if you're interested, read this section for more details.

CenterLine-C++ stores the files it needs for template instantiation in the template **repository**. The repository is a directory that can contain a **.o** file, which is the template instantiation object file, along with a **.c** (instantiation source file), a **.cs** (checksum file), a **.he** file (contains information about header file dependencies—see page 83) and the default name mapping file, **defmap**.

By default, the repository is created in the current directory and called **ptrepository**. (You can specify a different name and location with the **-ptr** switch.)

The name mapping file contains information about templates, including the names of the files where the templates are declared and instantiated. See the "Map files" section on page 75 for more information. You may find it helpful to look at the contents of each of the files created in the repository directory before you read the following discussion of the instantiation process.

The following steps summarize the instantiation process. For a more detailed explanation, see "Template Instantiation in C++ Release 3.0.2", an excerpt from the *AT&T C++ Language System Selected Readings*, which is available online as described on page 58.

1. At compile time, references to templates are compiled normally into external unresolved symbols, but nothing is instantiated. **ptcomp** logs every template that is declared in the default name mapping file, **defmap**. Every **class**, **union**, **struct**, or **enum** that is declared is also logged. The entry includes the type name and the basename (not the full pathname) of the file in which the declaration appears.

2. At link time, **ptlink** looks at all the files and archives, and the current repository, to determine whether there are any referenced template symbols that are unresolved. It checks header dependency (**.he**) files to make sure that any instantiations in the repository that are out of date are not used. If there are no unresolved symbols, the link is made.

3 If there are unresolved symbols, **ptlink** builds a list of class templates and function templates that must be instantiated. For each template, **ptlink** looks at the name mapping file to find the template declaration and definition files and the argument declaration files for all of the template arguments. The **-I** path that was passed to the compiler is used to find each of the files.

   The template definition file (that is, the file containing the implementation of the template) is assumed to be a file with the same name as the template declaration file, except that it has a different suffix. The mechanism for looking up the template types is described on page 74.

   If **ptlink** can't find any of the files, it issues an error message and the link fails.

4 **ptlink** uses the template declaration file, the template definition file, and the argument declaration files to build a temporary instantiation file. It then calls the C++ compiler to instantiate the template. The compiler only generates code for a specified list of symbols. If this is a class template, only the members that were needed are instantiated. The resulting object file is added to the repository.

5 If more template classes or functions need to be instantiated, Steps 3 and 4 are repeated.

6 When all the necessary templates have been instantiated, **ptlink** checks whether any of the new instantiations refer to new symbols and goes back to Step 2 if necessary.

7 Finally, **ptlink** produces a set of object files containing the instantiations that are passed to the actual link step.

See Figure 5 for a simplified conceptual illustration of the instantiation process.

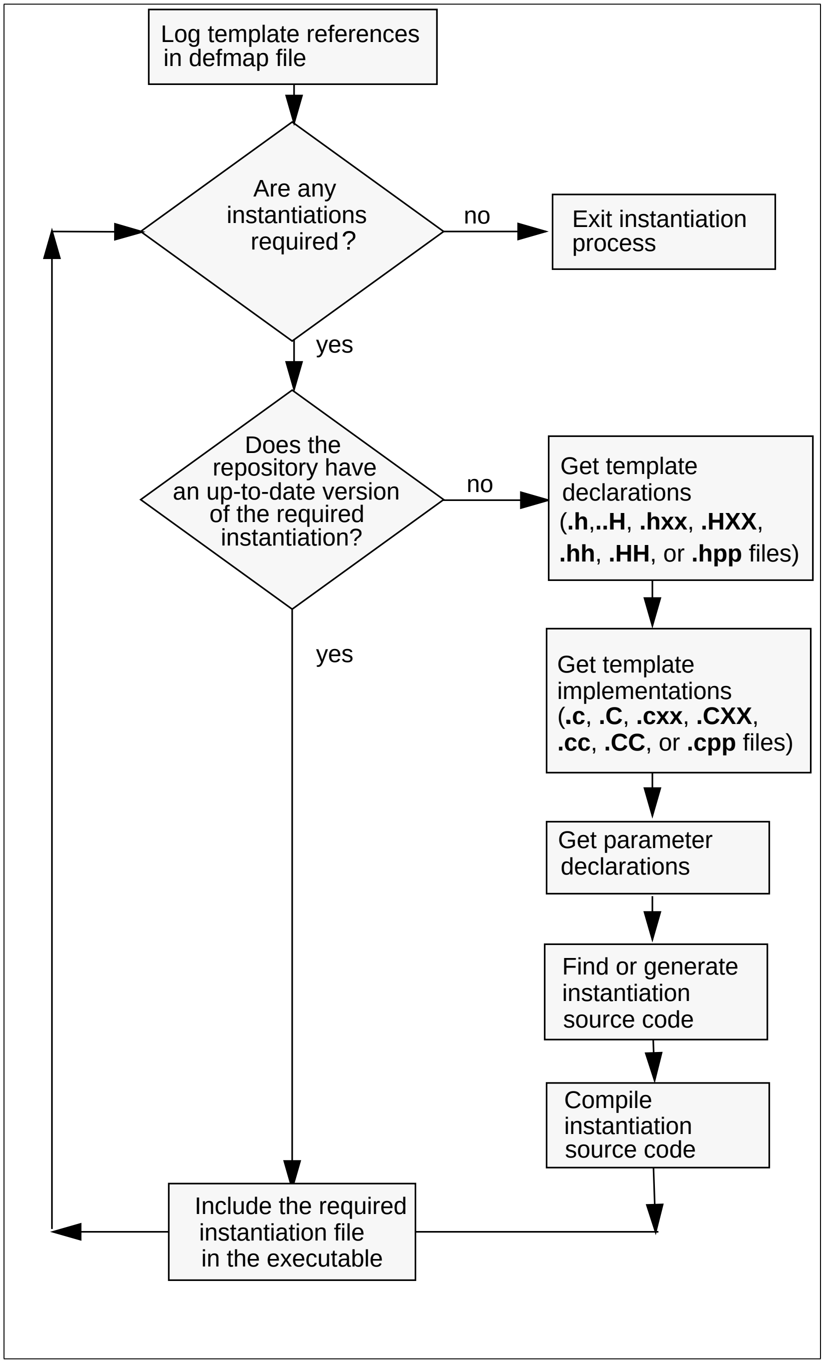


Figure 5 Steps in the Template Instantiation Process

Coding conventions

By default, CenterLine-C++ uses certain conventions for the structure of application files that use templates; they are the same as the conventions used by the AT&T C++ language system on which CenterLine-C++ is based.

Argument declaration files

By convention, an argument declaration file is used to declare types used as arguments to a template. For example, A and B are the argument types for the template class **Vector<A,B**>**. Fundamental types, such as **int** or **unsigned short***, are defined by the language and require no special declarations.

An argument type should be declared in a single header file that is self-contained or that includes other headers that it needs. If this is not possible then you must write a map file; see the "Map files" section on page 75.

You can define more than one type in the same header. An example of a self-contained header would be:

```
#ifndef INCL_A
#define INCL_A
class A {
  int x;
public:
  void f() {}
  void g() {}
};
#endif
```

while one with other **#include** directives might look like the following:

```
#ifndef INCL_B
#define INCL_B
#include "Point.h"
class B{
   Point p[10];
public:
   void rotate(int);
};
#endif
```

Include guards

INCL_A and **INCL_B** are **include guards**, used to prevent the same file from being included more than once. We recommend that you use include guards when writing template header files.

The compiler extracts type information from headers and remembers it so that the instantiation process can get it back when needed. If you use a template with arguments that are not fundamental types and have not been declared in an argument declaration file, the arguments can only be pointer or reference types, for example **Vector<A*, B&>**.

Template declaration files

Use a template declaration file to declare a template. Its structure mirrors that of a class declaration. For example, a declaration file could contain:

```
template <class T> class AAA {
    T x;
    int y;
public:
    void f();
    void g(T&);
};
```

For function templates, use a forward declaration:

```
template <class T> void sort(T*, int n);
```

When it finds a forward declaration, CenterLine-C++ creates a map file entry. This map file entry tells the instantiation system that an unresolved symbol might represent a template that needs to be expanded.

Like argument declaration files, a template declaration file should **#include** any header files it needs for the types it uses. However, header files for types used as template arguments or the definition of the template itself should not be included, since these are handled automatically by CenterLine-C++'s instantiation system. This means you should **#include** only the template declaration file (**Vector.h** in our earlier example) in the application file; do not include the definition file (**Vector.c** in our example) in the application. Also, do not include the template declaration file in the template definition file.

Template definition files

A template definition file contains the implementation of a template: the definitions of member function templates and initializers for static members of the template. The definition file has the same name as the template declaration file, but with a different suffix. See “Dynamic extension lookup” on page 73 for a list of suffixes.

If the template declaration from the previous section is in **AAA.h**:

```
template <class T> class AAA {
   T x;
   int y;
public:
   void f();
   void g(T&);
};
```

Then the definition file, **AAA.c**, would be as follows:

```
template <class T> void AAA<T>::f() { /* ... */ }
template <class T> void AAA<T>::g(T&) { /* ... */ }
```

In general, a definition file should not include the declaration file that matches it, nor the argument declaration files that declare any template argument types, unless you use include guards consistently as recommended on page 71. Including a guarded template definition file in a template declaration file will cause the definition file to be typechecked at application compile time, at the expense of slower compiling.

There must be a definition file for every declaration file, or a map file that overrides the standard convention. User-defined map files are described in “Specifying user-defined map files” on page 77. If a template definition file does not exist along the **-I** path, CenterLine-C++ issues a warning and does not include the file. All other missing files will cause a preprocessor error at instantiation time.

NOTE Use a definition file only to define templates in the corresponding declaration file. Information not related to template data or function definitions could be unnecessarily duplicated as part of the instantiation process for the templates, and therefore cause duplicate symbol errors when linking.

Inline functions

Inline template member functions are treated similarly to their class counterparts, except that they must currently be defined in the template declaration as shown in this example: they cannot be defined separately in the definition file.

```
template <class T> struct A {
void f() { /* ... */ }
};
```

If they are defined outside of the class body, but within the declaration file, inline template member functions will not be expanded inline. Instead, CenterLine-C++ generates and calls a static function.

Lookup schemes

In this section we summarize the schemes used for type and file lookup.

Type lookup

When **ptlink** does instantiation, it first makes a list of all the types used in the template class arguments. For example, if you have a function like this:

```
A<B,C>::func(D,E)
```

ptlink adds the types A, B, and C to the list and retrieves their declaration and implementation files. D and E are not added to the list. For function templates, the type name added to the list is the function name without arguments.

Dynamic extension lookup

In the examples we've used so far in this chapter, implementations of templates have been stored in template definition files with the **.c** suffix, and template declaration files have had a **.h** suffix. When you use other suffixes, **ptlink** must somehow determine what file to include in the instantiation file.

There are two kinds of file lookup: finding the header files that describe template arguments and finding the template types themselves.

Finding template parameters

The instantiation system looks for header files that describe template parameters (argument declaration files) in the map file. If it finds a header file for the type in the map file, it uses it. If not, it generates a forward declaration for the type in the instantiation file.

Finding template types

To find template types, the instantiation system must locate both the declaration and definition (implementation) files. To do this it uses the following procedure:

1. If there are both **@dec** (for declaration) and **@def** (for definition, or implementation) entries in the map file they are used.
2. If there is exactly one of **@dec** and **@def** in the map file, it is used to supply the basename, and then the **-I** settings are iterated over as an outer loop, and one of the following is used as the inner loop; either

   ```
   {".h", ".H", ".hxx", ".HXX", ".hh", ".HH", ".hpp"}
   ```

   if the declaration file isn't in the map file, or

   ```
   {".c", ".C", ".cxx", ".CXX", ".cc", ".CC", ".cpp"}
   ```

   if the definition file isn't in the map file. The first file that is found is used. This algorithm means that **ptlink** will attempt to exhaust all extensions in each **-I** directory before moving to the next. If no file is found **ptlink** goes to the next step.

   The list of suffixes is set by CenterLine-C++ to the default values shown above, or you can set them to the values you choose using the PTHDR and PTSRC environment variables. For example:

   ```
   export PTHDR=".h,.H" (SysV)

   setenv PTHDR ".h,.H" (BSD)
   ```

   ptlink ignores any item in the set of suffixes and issues a warning if it does not begin with a dot or has more than four total characters.
3. If there are no **@dec** nor **@def** entries in the map file, then the file basename for a template type **T** will be **T**. The algorithm in Step 2 is applied independently to get the declaration and definition file names. If **ptlink** cannot find either the definition or the declaration file names, it issues a warning and does not include a header file in the instantiation file.

Map files

Whenever you compile a source file that uses templates, CenterLine-C++ creates or updates a name mapping file in the repository. A map file contains mappings from type and template names to the source files that contain them. A map file entry is the only way that CenterLine-C++ can determine if an unresolved symbol might represent a template needing expansion.

The default map file

The default name for the current name mapping file is **defmap**; the preceding version is named **defmap.old**. This map file is maintained by CenterLine-C++, and you should not edit it. You can override the **defmap** file by specifying a user-defined name mapping file, as described on page 77.

Map files for the Vector example

Here's a portion of the **defmap** file after we linked the Vector example:

```
@tab
appVector
Vector__pt__2_i
@etab
...
@dec Vector @0 @1
"Vector.h"
...
```

The first few lines, bounded by **@tab** and **@etab**, are the string table, which is used by the instantiation system to compress the **defmap** file.

The entry beginning with **@dec** shows that type **Vector** is *declared* in **Vector.h**. If you look at the whole file, you will see that there is no **@def** entry specifying where **Vector** is *defined* (implemented). The instantiation system uses the algorithm on page 74 to locate the implementation for **Vector** in **Vector.c**.

In these lines:

```
@dec Vector @0 @1
"Vector.h"
```

@0 refers to the first item in the string table, **appVector**, and **@1** to the second item, **Vector__pt__2_i**, which is the name-mangled form of the name of the template class, **Vector<int>**. The lines indicate that **Vector** is declared in **Vector.h** and the type is valid for **appVector** (in both source and object form) and the template class **Vector<int>**.

Sharing a repository

Application file names are recorded in the name mapping file to handle the case where distinct applications share a repository. For example, suppose the Vector application and a banking application shared a repository. The string table at the top of the shared name mapping file might look like the following:

```
@tab
banking
account
appVector
Vector__pt__2_i
longVector
Vector__pt__2_l
@etab
```

Here’s the demangled version of this string table, which has six items.

```
$ c++filt defmap
@tab
banking
account
appVector
Vector<int>
longVector
Vector<long>
@etab
```

The string table is followed by lines that look like this:

```
@dec List @0
"List,h"
@dec account @0 @1
"account.h"
@dec Vector @2 @3 @4 @5
"Vector.h"
```

These **@dec** lines indicate that the **List** type is valid for the **banking** application, the **account** type for both the **banking** and **account** applications, and the **Vector** type for the **appVector** and **longVector** applications and also the **Vector<int>** and **Vector<long>** template classes.

Encoding of functions in map files

In map files, operator function templates are encoded as described in Section 7.2.1c in *The Annotated C++ Reference Manual* (see “Suggested reading” on page vi for publishing details). For example, operator << is encoded as **__ls**. Also, function template types are recorded without parameter information; as a result they appear as a single map file entry.

Specifying user-defined map files

You can specify additional name mapping files by naming them **nmap***name* and placing them in the repository; user-specified files take precedence over the default files and are considered in alphabetical order. For example, suppose you create your own map files, **nmap001** and **nmap2**; CenterLine-C++ looks at **nmap001** before **nmap2** before **defmap**.

You can create user-specified map files to override the lookup mechanism described in “Lookup schemes” on page 73.

Example of overriding default filenames

For instance, suppose you wanted to use the implementation of **Vector** that’s in **Vector.newc** as your “standard” definition, rather than the code in **Vector.c**. Then you could create a new mapping file, naming it **nmap1**, for instance, adding the specification for **@def**:

nmap1:

```
@def Vector
"Vector.newc"
```

Specifying application files in map files

Note that a map file entry does not have to specify application files; an entry without any applications serves as a last resort if the type cannot otherwise be found. Typically, application files are recorded to handle the case where there are distinct applications sharing one repository. Also, you don’t have to use **@0**, **@1** and so on as shown in “Sharing a repository” on page 76; you can spell the application name out. For example, instead of this:

```
@dec account @0 @1
```

you can write this:

```
@dec account banking account
```

Switches for templates

Table 6 describes the switches used by the CenterLine-C++ template instantiation system.

Table 6 Template Instantiation Switches

Name of Switch	What the Switch Does
-pta	Directs CenterLine-C++ to instantiate the whole template, rather than only those members that are needed.

-ptd*pathname*	Dumps list of instantiation objects to a file if any were recompiled or if the file does not exist. Also bypasses actual link step. Can be used with **-pti** in makefiles of this form:
	
	`appl:  appl.o ilist`
	`       CC -pti -o appl ‘cat ilist‘ appl.o`
	`appl.o: appl.c Vector.h A.h C.h`
	`        CC -c appl.c`
	`ilist:  always`
	`        CC -ptdilist appl.o`
	`always:`

-ptf	Forces **CC** to instantiate templates when the source file is compiled, instead of later, even if the program consists of more than one file. We do not recommend that you use this switch with applications that comprise more than one file.
-pth	Forces repository names to be less than 14 characters even if the operating system supports long names. This is useful in building archive libraries.
-pti	Ignores **ptlink** pass.
-ptk	Forces **ptlink** to continue trying to instantiate even after instantiation errors on previous template classes.
-ptm*pathname*	Have **ptlink** dump out a “link map” showing what actions the link simulator took.

Table 6 Template Instantiation Switches (Continued)

Name of Switch	What the Switch Does
-**ptn**	Changes the behavior of one-file programs to work like multi-file programs. If you do not set this switch for a one-file program, then by default, all templates are instantiated. See "Simple programs" on page 80 for more information.
-pto*pathname*	Consider instantiation modules in *pathname* to be out of date, and regenerate and compile. No checking is performed.
-ptr*pathname*	Specifies *pathname* as a repository. By default, *pathname* is **./ptrepository**. You can specify more than one repository by using the switch more than once; use the switch for each repository. If multiple repositories are specified, only the first is writable; the others are used to retrieve instantiation modules rather than store them as written. For example, **-ptr** might refer to a central project directory or a class library repository.
-pts	Splits instantiations into separate object files, with one function per object (including overloaded functions), and all class static data and virtual functions grouped into a single object.
-ptt	Use timestamps to determine when instantiations must be compiled. This switch is on by default.
-ptv	Specifies verbose mode for template instantiation; CenterLine-C++ announces each step in the instantiation process. This is especially useful when you're learning about templates.

Usage scenarios

This section describes various types of projects and the instantiation schemes that correspond to each.

Simple programs

By default, a one-file program that is to be compiled and linked causes CenterLine-C++to instantiate everything it can into the object file for the program. This means that the link-time instantiation system is bypassed, if all the templates and parameter types are found within the program itself. This behavior can be disabled using the **-ptn** switch.

Small and medium projects

By a small project we mean a project that has one programmer and uses one directory. Suppose that such a project needs some templates from a directory of template headers named **/usr/template/incl**. You could issue the following commands to accomplish this:

```
% CC -I/usr/template/incl -c file1.c
```

and at link time:

```
% CC -I/usr/template/incl file1.o file2.o -o prog
```

The repository used in this example would be the default, **ptrepository**.

If there is more than one project in a directory, it is better to use an explicitly named repository as a means of better separating one project from another. For instance, the following commands establish **rep1** as a user-specified repository:

```
% mkdir rep1
% CC -I/usr/template/incl -ptrrep1 file1.c
```

Repository permissions

When CenterLine-C++ creates the default repository, it gives it the same permissions as its parent directory, and files that are created in the repository have that same access.

This means that, if you want a repository to be shareable, you might have to change its permissions using **chmod**:

```
% chmod 775 ptrepository
```

Alternatively, you can create the repository in a directory with the desired permissions.

CenterLine-C++ deletes files in the repository before rewriting them, so if a repository has files in it and then the repository's permissions are changed, no access problems will come up.

Another approach is for team members to set the default creation mask at the shell level:

```
%sh umask 002
```

Large projects and multiple repositories

A large project often has a centralized set of source, library, and object files along with a local work area for each programmer. The best model for this kind of project is the use of multiple repositories. CenterLine-C++'s instantiation system looks first in your local repository and then the central one, both for map files and instantiation objects.

With such a scheme, you might issue a command such as the following:

```
% CC -I/usr/jones/tincl -I/usr/proj/tincl\
-I/usr/jones/incl -I/usr/proj/incl\
-ptr/usr/jones/rep -ptr/usr/proj/rep file.c
```

Given the preceding command, when it instantiates templates used in **file.c**, CenterLine-C++ uses the following repositories:

- **/usr/jones/rep** (to write instantiation modules as well as retrieve them)
- **/usr/proj/rep** (to retrieve existing instantiation modules)

Repository management

CenterLine-C++'s instantiation system adds to the repository but does not delete from it (except when it rewrites files). You may want to monitor the size of repositories periodically and delete obsolete files and repositories.

Sharing code and using archives

Instantiations in a repository are simply object files; you can easily export them into an archive. For example, with the default repository one can say:

```
$ ar cr projlib.a ./ptrepository/*.o
```

Such an archive may or may not be useful to other projects. By default, the system instantiates only what an application needs, and thus the object files will not contain all members of template classes. Another project with different needs might not be able to use such objects.

You can solve this problem by using the **-pta** switch, which tells CenterLine-C++ to instantiate everything; however, this solution wastes binary size. A reasonable strategy might be to use **-pta** initially and turn it off later in a project cycle.

You can also use the **-pts** switch to split up instantiations into separate object files for each function. This reduces problems resulting from object files clashing because they contain different but overlapping subsets of symbols.

Libraries

By library we mean a collection of object files, also known as an archive. Suppose you have a library that uses templates, but end users of the library do not know or care about templates. You can avoid the instantiation process for those users by **forming the closure** of the library; forming closure means instantiating everything into object files and adding the objects to the library.

For example, if you wanted to form closure for a library named **/usr/proj/lib.a**, you could say:

```
$ mkdir scratch
$ cd scratch
$ ar x /usr/proj/lib.a
$ CC -I/usr/proj/tincl -I/usr/proj/incl -pts *.o
$ rm -f /usr/proj/lib.a
$ ar cr /usr/proj/lib.a *.o ./repository/*.o
```

Use the **-pts** switch with **CC** to split the instantiations into separate files.

When you follow the preceding example, you may get a link error that you can ignore; it occurs because the code does not have a **main()**.

Link-simulation algorithm

The **ptlink** link-simulation mechanism is designed to support archive libraries with partially-instantiated template classes in them, by using functions found in libraries whenever possible. The algorithm is as follows:

1. Standalone objects are always “linked,” and objects encountered as the link simulator traverses the archive are linked if symbols from them are needed.
2. For each text. data, or bss[1] symbol in the library object to be added, **ptlink** checks to see if the symbol is already in the link simulator symbol table and if it is already defined to the correct type. If there are no symbols already defined, the object can be linked.
3. If one or more symbols is already defined, then each text, data, or bss symbol that was previously undefined is marked as undefined and undefinable. No future object can resolve the symbol. This step is necessary to preserve archive semantics.

Note that object filenames in the repository may be longer than the 14 characters that **ar** will handle. You can use the **-pth** option to limit names to 14 characters when you compile, or rename object files; a tool for this purpose is described in the "Tools" section on page 95.

Dependency checking

The template instantiation system has the following scheme for checking whether instantiated objects are out-of-date.

CenterLine-C++ compares the timestamps of **#include** declaration files in the instantiation file with the timestamp of the instantiation object. To handle nested **#include** directives, CenterLine-C++ creates a cache, which it stores in the repository with a **.he** extension. For example, the cache for the Vector example is in **Vector__pt__2_i.he**.

The first line of the **.he** file shows the **-I** and **-D** switches used with **CC**. Subsequent lines contain the names of all header files. An object file is considered out-of-date if it is older than any of the headers on the list, or if the **-I** and **-D** switches have changed.

1. The bss section of an object file contains uninitialized data. See your system manual page for **nm** for more information.

Forcing reinstantiation

Sometimes it is desirable to get around dependency checking. To force reinstantiation, you can enter the following:

```
% touch template_name.suffix
```

where *template_name.suffix* is the name of the template definition file containing the implementation of the template you want to force to reinstantiate.

Alternatively, you can delete all object files in the repository; however, this works only if your makefile has an explicit dependency on the template instantiation file.

Compiling and linking

All switches that apply to the creation of an executable must be on the **CC** command line, whether they pertain specifically to compiling, linking, to both, or to the template instantiation process.

For instance, some of the switches that pertain to linking are as follows:

-L*library_dir* **-l***libname* **-o** *executable_file_name*

Some switches related to compiling are as follows:

-D*NAME1* **-I** */ header / file / directory* **-U***NAME2* **-O**

Some switches are used by both the linking and compiling phases; typically these switches relate to debugging:

-g +d

Using CC with templates in header files

Template instantiation with **CC** combines compiling and linking into one operation. When you compile source files that refer to templates, you do something like this:

```
% CC -g +d -DTHIS -DTHAT -I/some/directory -c app.C
```

In this example, all the header files are found if they exist in the current directory or in **/some/directory**.

But these header file directories and macro definitions are not saved with the object module. So your code will generate unresolved template references if your templates depend on header files in **/some/directory** and you try to do the following to link:

```
% CC -g -o app app.o ...
```

The original switches used in compiling **app.o** are not available. The only flags available when compiling template instantiations are those given on the **CC** command line when you link.

In this case, to make header files work with **CC** you must do something like the following:

```
% CC -g -DTHIS -DTHAT -I/some/directory app.o ...
```

Using automatic tools and make

In some cases you may want to separate instantiation from the linking phase. You can do this using the **-ptd** switch, which performs instantiation without linking, and the **-pti** switch, which bypasses the instantiation step. Part of a makefile that uses this construction is shown in the **-ptd** entry in Table 6, "Template Instantiation Switches," on page 78.

When you compile an application that uses templates, you need to specify the file that contains the template implementation (the template definition file) as a dependency in your makefile. And, if you use automatic tools that check for dependencies, you have to manually indicate that template definition files are dependencies.

Specializations

A specialization is a means of overriding the standard version of a template class or a particular member of the class. Typically you use specializations to improve performance, or to reuse most of the code for a given template while providing your own version of a particular member function.

To use a specialization, first compile the source file containing the specialization with the **-c** switch, then link the resulting **.o** file with your application.

A specialization example

For instance, suppose you want to use the **appVector.C** application described in “Using templates with CenterLine-C++” on page 63. However, you want to override the implementation code in **Vector.c** for the case of integers as the parameterized type.

Here’s the template implementation in **Vector.c** as we described it earlier:

```
template <class T> Vector<T>::Vector()
{
   size = 3;
   data = new T[size];
}
```

In this case, suppose the source code for the specialization is in a file named **spec_vec.c**:

```
#include <iostream.h>
#include "Vector.h"
Vector<int>::Vector()
{
 size = 3;
 data = new int[size];
 // add initializer and output for specialization
   for (int i=0; i<size; i++) data[i]=0;
cout<< "this is a specialization for Vector" << endl;
}
```

Notice that the specialization does not contain template **<class T>**. Also, we modified the definition of the constructor by adding a **for** loop initializing the array along with a call to **cout**.

For convenience, here’s the application, as we described it earlier:

```
#include <iostream.h>
#include "Vector.h"

main()
{
Vector<int> v;
int i;

// put data into vector
for (i = 1; i <= 5; i++) v[i] = i * i;

// display data in vector
for (i = 1; i <= 5; i++)
     cout << i << " " << v[i] << "\n";
}
```

To compile and link this application with the specialization:

```
% CC -c spec_vec.c
% CC appVector.C spec_vec.o
```

The compiled specialization must be placed on the link line to prevent the general versions from being instantiated at link time.

The program's output is as follows:

```
% a.out
this is a specialization for Vector
1 1
2 4
3 9
4 16
5 25
```

Static template class data members

Specialization of static template class data members is done in a similar way. For instance, a template declaration such as the following provides a general template initializer:

```
template <class T> int A<T>::x = 97;
```

To specialize this, you could say:

```
int A<int>::x =52;
```

somewhere in the application.

Examples

This section describes a few more small sample cases.

Single file

In the simplest case, the template definition and the application code that uses it are all in the same file, **userapp.C**:

```
#include "String.h"
template <class T> class Stack {
   T* head;
public:
 Stack() : head(0) {}
 T pop();
 void push(T&);
};
```

```
template <class T>
T Stack<T>::pop()
{ /* ... */ }

template <class T>
void Stack<T>::push(T& arg)
{ /* ... */ }

main()
{
Stack<String> s;
/* Code that uses push and pop */
}
```

To execute this code in CenterLine-C++, do the following:

```
% CC userapp.C
```

In this case, the instantiation is completely automatic; you need do nothing further to instantiate the **Stack** class template used in **main()**.

When **userapp.C** is compiled, the **push** and **pop** references are compiled as normal function calls. No reference to **Stack<String>::Stack()** is generated because it is inline. The name mapping file is updated to show the declaration of templates and classes:

```
@dec String userapp
"String.h"
@dec Stack userapp
"userapp.c"
```

Separate compilation

The next example is more typical than the preceding one. The template is declared in a declaration file (**Stack.h**), the implementations are provided in a separate definition file (**Stack.c**), and the application is in a third file (**userapp.C**):

Stack.h:

```
template <class T> class Stack
{
T* head;
public:
 Stack() : head(0) {}
 T pop();
 void push(T&);
};
```

Stack.c:

```
template <class T>
T Stack<T>::pop()
{ /* ... */ }

template <class T>
void Stack<T>::push(T& arg)
{ /* ... */ }
```

userapp.C:

```
#include "String.h"
#include "Stack.h"

main()
{
Stack<String> s;
/* Code that uses push and pop */ }
```

Here, the scenario is the same as in the preceding example, except that CenterLine-C++ gets the template declaration and definition from different files — **Stack.h** and **Stack.c**, instead of **userapp.C**. Keep in mind that **Stack.c** must be available along the **-I** path in order for the instantiation to succeed.

NOTE	Given the correct setup of files and **-I** path, the instantiation process in all these examples is automatic. The following paragraphs describe the details of what goes on “behind the scenes” in the last example.

Here are the implementation details for the last example:

1. When you compile **userapp.C**, the references to **Stack<String>::push(String&)** and **Stack<String>::pop()** are considered normal function calls. Since **Stack<String>::Stack()** is inline, no reference to that function is generated.

2. CenterLine-C++ determines that the following functions must be instantiated:

```
Stack<String>::push(String&)
Stack<String>::pop()
```

3. CenterLine-C++ checks the repository for a file that contains these instantiations. If there is one that is up-to-date, CenterLine-C++ adds that file to the list of files to be linked and compiled, if necessary, and goes on to Step 5.

4 If the repository does not contain an up-to-date file with these instantiations, they are instantiated. Both members of **Stack<String>** will be instantiated into the same source instantiation file.

According to the **defmap**, the template declaration file is **Stack.h**. The template definition file has the same name as the template declaration file, except that the suffix is changed to **.c**, so, in this case, the template definition file is **Stack.c**.

Also, according to the **defmap**, the parameter declaration file is **String.h**.

CenterLine-C++ instantiates **Stack<String>** by building an instantiation source file that contains the definitions of **Stack<String>::push(String&)** and **Stack<String>::pop()**, plus any virtual functions in **Stack<String>**.

5 CenterLine-C++ compiles the instantiation source file, if necessary, and stores the resulting object file in the repository.

6 CenterLine-C++ checks for any further new instantiations needed; if there are, CenterLine-C++ repeats the preceding process, starting with Step 2.

7 If CenterLine-C++ is satisfied that all required instantiation files are available, it calls the linker to complete the link.

Specialization at link time

It is legal for a special case of a template member to be discovered at link time. For example, given the files shown in "Separate compilation" on page 88, suppose this additional file were provided at link time:

stringpop.c:

```
#include "String.h"
#include "Stack.h"
/* Special case version of Stack<String>::pop */

void Stack<String>::pop()
{ /* ... */ }
```

This implementation of **Stack<String>::pop()** is used instead of the one in the template definition file, **Stack.c**, so CenterLine-C++ determines that only **Stack<String>::push(String&)** needs to be instantiated.

Avoiding the most common pitfalls when using templates

Templates are probably easier to use than most people expect; once you set up your files correctly, the entire process can be handled automatically by CenterLine-C++.

Here we reiterate a few points made earlier in the section on templates; these tips might help you avoid some mistakes often made by new users of templates.

- Do not compile the file containing your template implementation (the template definition file), and do not include it in your application file. Doing so interferes with CenterLine-C++'s automatic instantiation process.
- Use default naming conventions for your files, that is **.h** and **.c** or **.H** and **.C** for declaration / definition file pairs, unless you need to use other suffixes.
- Do not include a template declaration file in the template definition file, unless you use include guards, and do not use the template definition file to define anything except templates.
- Use include guards to prevent redundant compilation of declaration (header) files.
- Do not edit the **defmap** or any instantiation files generated by CenterLine-C++. If necessary, create an **nmap** file to override the default rules for finding template files.
- Do not specify any files to be included in the repository to the linker explicitly; allow the automatic instantiation process to do any linking related to templates.
- Keep in mind that you might get syntax errors during the final linking phase, since templates are instantiated later. If you do get syntax errors at the instantiation phase, edit only the template definition file, not your application file.

  For instance, if you get an error message indicating "missing template arguments", you may have forgotten to specify the parameter for a template in your template definition file. This could be a matter of simply remembering to write <T> after the template name.

Troubleshooting

This section is based on information from AT&T about **cfront**. It describes possible difficulties you might encounter.

Network timestamps

If you have a network of workstations, timestamps may not be synchronized, in which case dependency checking will not work correctly. This problem must be solved by system administration.

External name length limitations

Symbol names in object file symbol tables must fully describe the template class used by a given function or data item. The instantiation process cannot resolve symbol names correctly if the system imposes a name length limit of 8 or 32 characters. Using a **typedef** to shorten a long name will not solve this problem because the **typedef** name is expanded to the underlying types when external names are encoded.

Map file problems

If you have many programs in the same directory using the same type name, for example, test cases using the type T, the default map file will become very large. You can compress the file by using a string table, as described in "Sharing a repository" on page 76.

Some out-of-date information is deleted when a file is recompiled, but some garbage slowly accumulates in map files.

Violation of the one definition rule

Because of separate compilation, the C++ compiler will accept usage such as:

```
// file 1 struct A {};
// file 2 template <class T> struct A {};
```

even though this violates the rule that there must be only one definition of each object used in a program. Because type mapping information is collected into one file, the instantiation system will catch many such errors. The form of the error is:

```
fatal error: type A defined twice in map files
```

Picking up the wrong versions of headers

Some source code control and configuration management systems support named versions of source files and headers, and program compilation is done with particular sets of versions of files (a configuration). Template instantiation does not cause any problems with this, but you must ensure that the same versions of files are specified via **-I** at link time as are given at compile time.

Replaying source files

If a source file looks like this:

```
// main.c
#include <Vector.h>
struct A {};
main()
{
Vector<A> a;
a.f();
}
```

and **Vector.h** does not have include guards, then it will be included twice, once to get at the type **Vector** and once as an indirect result of including **main.c** to get at the type A.

The workaround for this is either to use include guards or else completely define the types in header files or in **main.c**.

Function templates

A function template is encoded just like a C++ function. At instantiation time, there is no way to tell them apart. Therefore, the instantiation system tries to instantiate function templates only if an entry is found for them in the map files. This entry will not be there unless a forward declaration, such as

```
template <class T> void f(T);
```

has been seen.

Another problem occurs if only a function definition is given in a single-file application, and then **-ptn** or **-c** is used to tell the instantiation system not to instantiate:

```
template<class T> void f(T) {}
main()
{
f(37);
}

$ CC -ptn prog.c
```

Because there is no declaration, no entry is made in the map file, resulting in an unresolved global **f(int)** at link time. The workaround is to use a declaration or **-ptf**, or do not use **-ptn** for multi-file applications.

Specializations of function templates and parameter matching can present another problem. Given this function template:

```
template <class T> void f(Vector<T>&);
```

and this declaration of a specialization:

```
void f(char*);
```

If the specialization is not defined anywhere, the pre-linker will find it to be unresolved. The pre-linker will then look for **f** in the map files and find it, and attempt to instantiate the **f(Vector<T>&)** template with a **char*** argument.

Static data member initialization

The instantiation system considers that the tentative definition (global common) that the C++ compiler emits for each static data member of a template class represents an undefined external symbol that must be defined and initialized somewhere.

For example:

```
template <class T> struct A {
 static int x;
};
```

by itself would result in an unresolved external.

This usage follows the C++ standard, but the C++ compiler has not enforced it up to now. An initializer might look like this:

```
template <class T> int A<T>::x = 47;
```

or this:

```
int A<char*>::x =89;
```

The first of these is a general template initializer, the second a specialization.

Type checking of template members

By default, only members of a template class that are used are instantiated. Other members are not typechecked and therefore legally could contain errors.

All virtual functions are instantiated because there is no way to tell whether they are needed.

If you use the **-pta** or **-ptf** switch, CenterLine-C++ will try to instantiate all members of needed template classes, with potential errors.

Renaming object files

The basename of an object file is used to validate type entries in map files. If the name changes, the type entry will be invalid unless other object files specified along with the renamed one are also found on the basename list in the map file.

The simplest solution to this problem is to write a map file with a type entry with no list of basenames (see “Specifying application files in map files” on page 77).

Debug formats and large binaries

The instantiation system creates one object file for each template class. With some debug formats, the linker does not merge duplicated strings and other debug information occurring in several object files. This can cause a large increase in binary size. The problem has no easy solution.

Tools

This section describes tools provided as part of the AT&T C++ Language System that we include with CenterLine-C++. They are in the following locations:

CenterLine/clc++/*arch_os***/pt/tool1**

CenterLine/clc++/*arch_os***/pt/tool2**

Because the template instantiation repository is a UNIX directory and the files in it are not special in any way, it is possible to use standard utilities in various ways to get at information.

For example, consider a system that has only 14-character filenames. Hash codes are used to name files in place of complete mangled names, and it would be nice to come up with a correspondence list showing the mapping between hash codes and template names.

A shell script to do this is **tool1**:

```
#!/bin/sh

# display the template class for each instantiation
# file in the repository

PATH=/bin:/usr/bin:/usr/ucb

pn=‘basename $0‘
rep=$1
if [ "$rep" = "" -o ! -d "$rep" ]
then echo "usage: $pn repository" 1>&2
     exit 1
fi
cd $rep
ls *.c |
while read fn
do
    n=‘sed -n ’1s/^/\* ‹.*›\*/$/\1/p’ $fn‘
    echo "$fn --> $n"
done

exit 0
```

Another tool, **tool2**, can be used to package the object files in a repository into an archive, with renaming to short names for **ar**:

```
#!/bin/sh

# export contents of repository into an archive

PATH=/bin:/usr/bin:/usr/ucb

pn=‘basename $0‘

t=/tmp/$pn.$$
trap "rm -rf $t; exit 2" 1 2 3 15
rm -rf $t
mkdir $t

if [ $# -ne 2 -o ! -d "$1" ]
then
    echo "usage: $pn repository archive"
    exit 1
fi

n=1
for i in $1/*.o
```

```
do
    cp $i $t/${n}.o
    n=‘expr $n + 1‘
done

rm -f $2
ar cr $2 $t/*.o
if [ -x /bin/ranlib -o -x /usr/bin/ranlib ]
then
    ranlib $2
fi

rm -rf $t

exit 0
```

Summary of terminology

For your convenience, we summarize some of the terminology related to templates as used in CenterLine-C++. Terms are listed in alphabetical order.

argument declaration file
: A file containing the declaration of a **class**, **struct**, **union**, or **enum** type.

defmap
: The default name for the name mapping file.

header cache
: A header dependency file with the suffix **.he** in the repository, which is used to store the list of headers needed by each instantiation.

name mapping file
: A file in the repository that contains information needed to define and instantiate templates, including where each named type used in a template instance is declared.

repository
: A special directory created by CenterLine-C++ the first time a file containing a template declaration or a template instance is compiled. If an application does not use templates, then no repository is ever created. By default, this directory is created in the working directory and is called **ptrepository**.

specialization
: A user-supplied definition or implementation of a template class or function that overrides the default instantiation.

template declaration

A declaration of a class template or a function template. It starts with the keyword template.

```
// class template
template <type T> class Stack {member(T);...};

// function template
template <type T> void print(T);
```

template definition

A definition of the member functions and initializers for static data members of a class template, or of a function template.

```
// template member function definition
template <type T> class Stack<T>::member(T) { ...}

// template function definition
template <type T> print(T) { ... }
```

template definition file

A file that contains definitions (implementations) for some or all of the needed member functions of a class template, or the definition of a function template.

template instance

A specific instance of a template. It can be any of the following:

- A template class implicitly declared by using a template class name:

```
Stack<int> // template class
```

- A template function explicitly declared:

```
void print(int); // template function
```

- A template function implicitly declared by calling it or taking its address:

```
print(5); // also a template function
```

template instantiation

An automatically generated definition of of a template function instance, or of the member functions of a template class instance.

5 Introduction to the Debugger: A Tutorial

This chapter provides a tutorial for newcomers to the CenterLine-C++ debugger, **pdm**. *The tutorial guides you through a sample session that includes the following activities:*

- *Debugging basics*
- *Correcting compiler and make errors*
- *Debugging a corefile*

Debugging basics

The CenterLine-C++ debugger is a symbolic debugger for debugging fully linked executables. Although the CenterLine-C++ debugger is similar to debuggers like **gdb** and **dbx**, it provides a graphical user interface with an integrated set of graphical browsers for examining and debugging your code more efficiently.

The CenterLine-C++ debugger is also known as **pdm**, which stands for process debugging mode. Although CenterLine-C++ supports only this single mode of debugging, other CenterLine products support multiple debugging modes.

This chapter provides a tutorial to guide you through a sample debugging session. You learn how to correct compiler and make errors and debug a corefile. For detailed information about debugging tasks, refer to Chapter 6, “Debugging with CenterLine-C++,” on page 117.

Specifying your editor

There are many places in the debugger where you can invoke your editor. Before you start the debugger, you can use the **EDITOR** environment variable to specify either **vi** or **emacs** as the editor for the debugger to invoke. The debugger uses **vi** as the default unless you specify **emacs**.

To specify GNU Emacs as the editor for the debugger to invoke, use the following shell command before you start the debugger:

```
% setenv EDITOR emacs
```

If you specify **emacs**, note that the debugger supports only GNU **emacs**, and the version of GNU **emacs** must be capable of running as an X Window System client.

Setting up the Bounce program

To explore the debugger, the tutorial uses a simple program named Bounce. The Bounce program creates a new window and bounces a rectangle within the window. The existing program has two different problems that you will fix.

If you have not set up the CenterLine-C++ examples directory yet, refer to “Setting up the examples directory” on page 9 for instructions. Change to the examples directory:

```
% cd c++examples_dir
```

NOTE When you set up the examples directory, CenterLine-C++ modifies the makefile in it to include the absolute pathname to the **CC** and **clcc** commands on your system. Because of this, we recommend that you start the debugger from the *same host* where you set up the examples directory. Otherwise, you may not be able to run the tutorial successfully until you edit the CXX and CC variables in the makefile yourself.

The **c++examples_dir** directory contains files for several programs, including the Bounce program. Bounce consists of the following subset of files in **c++examples_dir**:

Makefile	**mainfixed.C**	**shapes.C**	**table.h**	**x_image.h**
link.C	**rect.C**	**shapes.h**	**x.C**	**link.h**
rect.h	**skip**	**x.C.orig**	**main.C**	**shapelst.C**
x.h	**main.C.orig**	**shapelst.h**	**table.c**	**xfixed.C**

NOTE Before you invoke the debugger, be sure to set up your **DISPLAY** environment variable according to usual X Window System conventions. For example, if your host is named **baxter**:

```
% setenv DISPLAY baxter:0
```

Starting the debugger

To start the debugger, use the **centerline-c++** command with a switch specifying the user interface you prefer:

```
% centerline-c++ -motif
```

Or

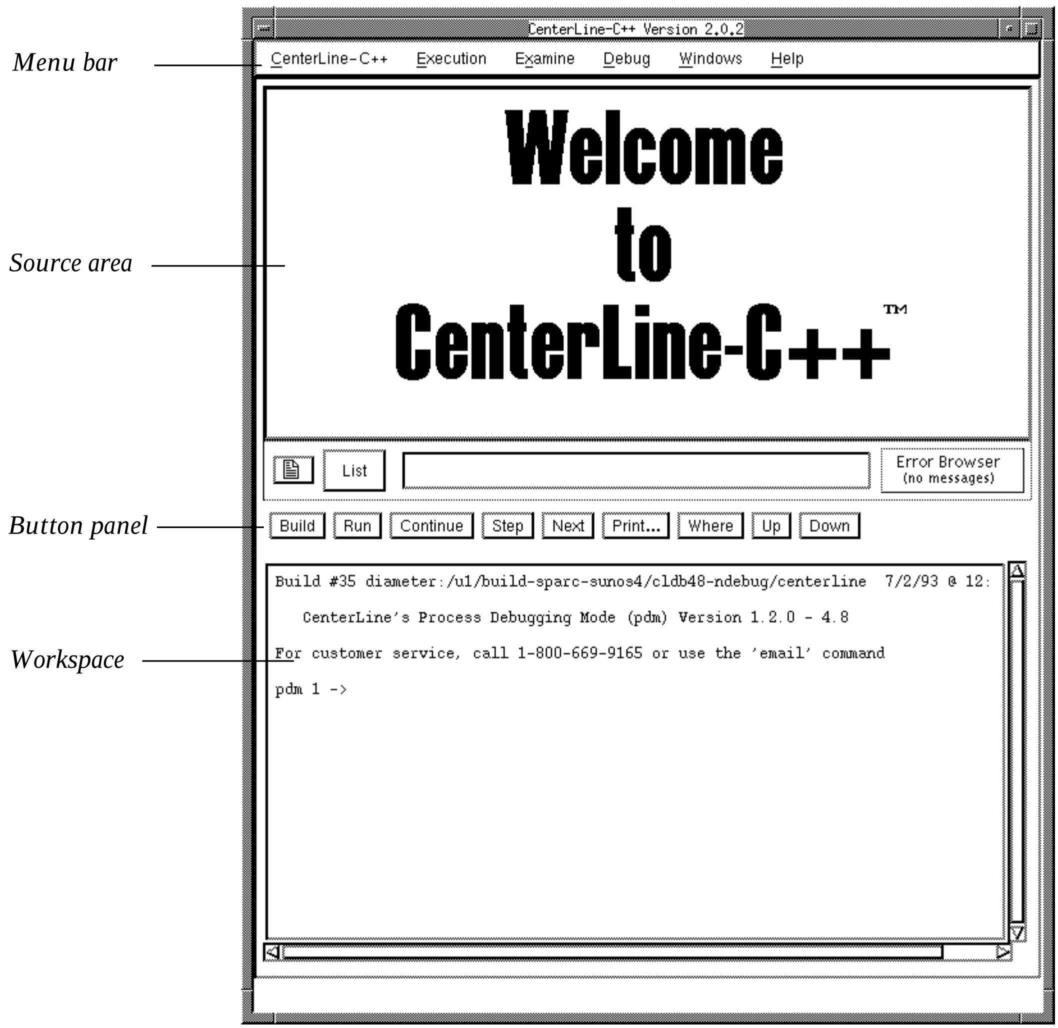
```
% centerline-c++ -openlook
```

If you omit the switch specifying Motif or OPEN LOOK, the debugger uses the default GUI, which is platform-specific. The default GUI for Sun SPARC machines is the OPEN LOOK GUI; for other architectures, the default is the Motif GUI. Illustrations in this tutorial show the Motif GUI.

The **centerline-c++** command is installed in a **CenterLine/bin** directory, which could be installed anywhere on your system. If **CenterLine/bin** is not in your path or if you need to know the absolute path for **CenterLine/bin**, see your system administrator.

The Main Window

When you start the debugger, it displays an icon and then a Welcome to CenterLine-C++ window. The debugger then opens the Main Window, as shown in the following illustration:



The Main Window provides a central work area and contains four main regions: the Menu bar, Source area, Button panel, and Workspace.

- In the Menu bar, you can select commands for executing and examining programs.
- In the Source area, you can list source code and manipulate debugging items. For example, you can set and delete breakpoints and actions in the Source area.
- In the Button panel, you can use buttons to execute and examine programs.
- In the Workspace, you can enter debugging commands and view the results. The Workspace is a command processor that allows you to enter debugging commands directly and to pass shell commands to a subshell.

In general, the same features are available either by using menus, buttons, and other graphical elements of the GUI or by entering commands directly in the Workspace. You can choose whichever means of access is most convenient for you. In this tutorial, we show you several different ways of entering the same command

The Main Window also acts as a hub for the debugger's graphical browsers. You can open these browsers with the **Windows** menu, and each of the browsers also has a similar **Windows** menu. You'll learn more about the browsers later in the tutorial.

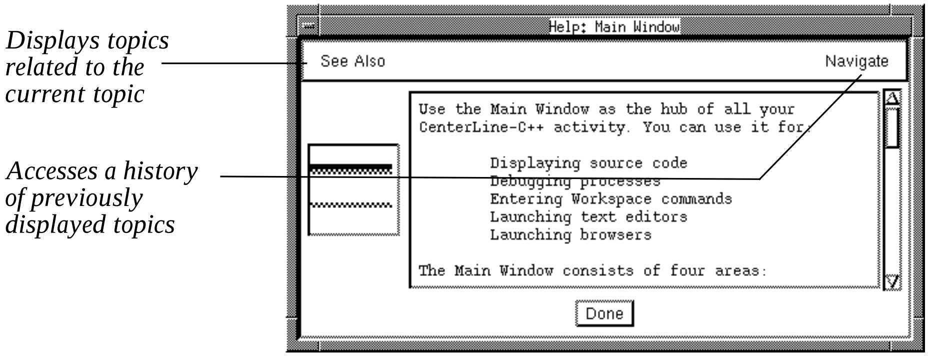
Getting help

The debugger offers multiple sources of help: context-sensitive help, a **Help** menu, Workspace help, and the Man Browser.

Using context-sensitive help

The debugger provides an extensive system of context-sensitive help to assist you in using the product. To get information on any graphical object, move the cursor over the item or region and press the F1 or Help key. A Help window appears describing the object.

For example, in the Main Window, if you move the mouse pointer between the menu bar and the Source area and press F1, you see the following help topic:



You can also explore the **See Also** and **Navigate** menus. To dismiss the Help window, select the **Done** button.

Using the Help menu

In addition to context-sensitive help, the debugger offers help on a range of topics. You can access this help through the **Help** menu in the Main Window or any of the browsers. For example, the **On Window** help topics gives an overview of each of the debugger's primary windows. Before moving on, you might want to spend some time familiarizing yourself with the topics covered in the **Help** menu.

Using Workspace help

In the Workspace, the **help** command displays quick usage information about debugging commands. For example, to get a usage summary for the **email** command, enter the following in the Workspace:

```
pdm -> help email
```

Using the Man Browser

For in-depth information on topics, the Man Browser displays reference information on each Workspace command as well as shell commands and X resources. You can open the Man Browser from any primary window by displaying the **Windows** menu and selecting **Man Browser**.

You can also open the Man Browser by typing this in the Workspace:

```
pdm -> man topic_name
```

Managing windows

If you have finished using a window, select the **Dismiss** or **Cancel** button to close the window. Resize a window or use the horizontal and vertical scroll bars to view any graphical code representation that is too large to fit in the window.

Using the Run Window

Use the Run Window to view the output from and enter the input to programs that you are running in the debugger. The Run Window, an **xterm** window, is the first window to appear (it is iconified) when you start up the debugger.

Quitting the debugger

You can exit from the debugger at any time by selecting **Quit CenterLine-C++** from the **CenterLine-C++** menu in the Main Window or entering **quit** at the Workspace prompt.

Correcting compiler and make errors

You can build and run programs from the debugger to take advantage of its Error Browser and seamless integration with your text editor. As your first step in exploring the Bounce program, you build it. To do so, use the UNIX **make** utility from the Workspace. As the debugger builds the Bounce program, it echoes the commands executed. For **CC**, the debugger echo the switches passed to the C++ translator:

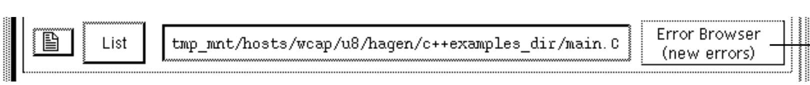
```
pdm 1 -> make tutorial
CC -I/usr/include/X11R4 -I/usr/include/X11R5 ... +d -g -c x.C
CC +C +d x.C:
"x.C", line 261: error: syntax error -- did you forget a ';'?
"x.C", line 261: error: MAX_ITERS's definition is nested (did you forget
a ''}''?)
"x.C", line 261: error: uninitialized const MAX_ITERS
"x.C", line 261: warning: result of / expression not used
"x.C", line 261: warning: MAX_ITERS used but not set
3 errors
*** Error code 1
make: Fatal error: Command failed for target 'x.o'
pdm 2 ->
```

NOTE If the **make** command cannot find the **CC** or **clcc** commands, check the values of the CXX and CC variables in the makefile and edit them as needed.

If the **make** command fails because your system doesn't have the X11 header files and libraries installed in the expected places, you need to edit the CL_INCS and CL_LIBS variables in the makefile.

Examining errors in the Error Browser

The Workspace output shows that the build results in four errors (three from the compiler and one from **make**) and two warnings. Notice that the Error Browser button in the Main Window indicates there are errors in the Error Browser..



Error Browser button

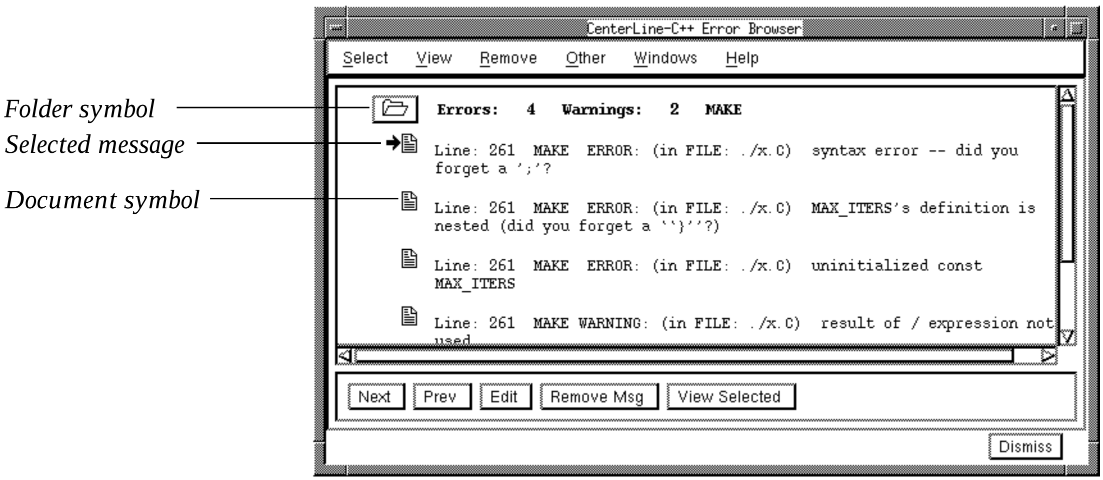
To examine these errors in more detail, open the Error Browser:

1. In the Main Window, click on the Error Browser button.
2. Position the Error Browser so you can see the Main Window at the same time.

The Error Browser displays a folder containing four **make** errors and two **make** warnings. To examine the errors:

In the Error Browser, click on the Folder symbol.

The Error Browser displays the error messages, as shown in the following illustration.



Editing the source code to fix the error

When you open a folder or select a message in the Error Browser, the Source area in the Main Window lists the source file causing the error (**x.C**) and indicates the line that caused the error (line 261). In this case, the line defines a symbolic constant, but it is missing an assignment operator (**=**).

In C++, you can replace many **#define** preprocessor directives with constants. For example:

```
#define MAX_ITERS 600/WIDTH /* C version */

const int MAX_ITERS = 600/WIDTH; // C++ Version
```

In this case, the assignment operator was omitted when the line was changed.

To invoke your editor on the source file:

> In the Error Browser, click on the Document symbol next to the first error.

Your editor opens **x.C** and positions the cursor at line 261. From your editor, do the following:

1. Add an **=** between MAX_ITERS and 600 / WIDTH.
2. Save the file.
3. Close the editor.

Building and running the fixed program

Now that you've fixed the error, go to the Main Window and rebuild the **tutorial** target from the Workspace.

```
pdm 2 -> make tutorial
```

The Error Browser button indicates that there are no messages, and the messages are cleared from the Error Browser. The Bounce program compiles without errors. You can dismiss the Error Browser:

> Select the **Dismiss** button.

Although the Bounce program compiles, it still has a problem that will generate a segmentation fault. To run the program and dump a corefile, use the following commands in a separate shell. (The corefile is about 8.5 MB in size.)

> If you use the C shell on a Sun platform, you might need to use this command to allow the whole corefile to be dumped.
>
> ```
> % unlimit coredumpsize
> ```
>
> To generate the corefile:
>
> ```
> % cd ~/c++examples_dir
> % tutorial
> Segmentation fault (core dumped)
> ```

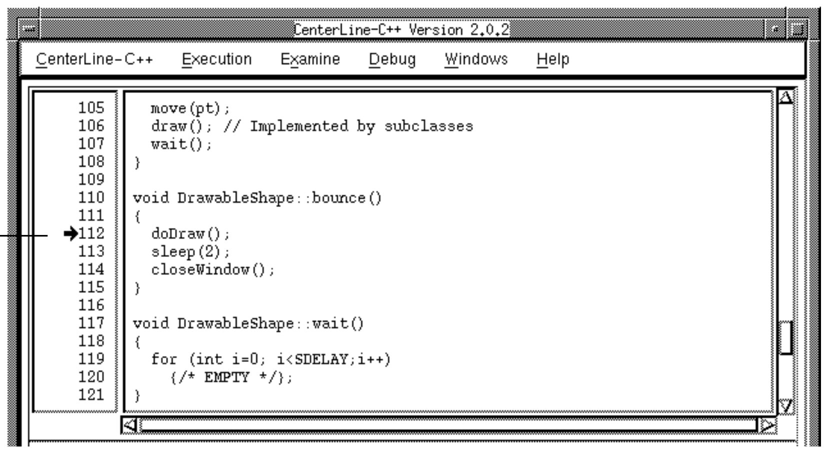
Debugging a corefile

You can debug the Bounce program and the corefile it dumped in the debugger. To do so, issue the following command in the Workspace:

```
pdm 3 -> debug tutorial core
Debugging program 'tutorial'
Core was generated by 'tutorial'.
Program was terminated with signal 11, Segmentation fault.
#0  0x326c in DrawableShape::bounce (this=0x0) at shapes.C:112
    112   doDraw();
pdm 4 ->
```

The Workspace indicates the line of source code that generated the segmentation fault. The Source area displays **shapes.C** and uses the Execution symbol to indicate where execution stopped:

Execution symbol

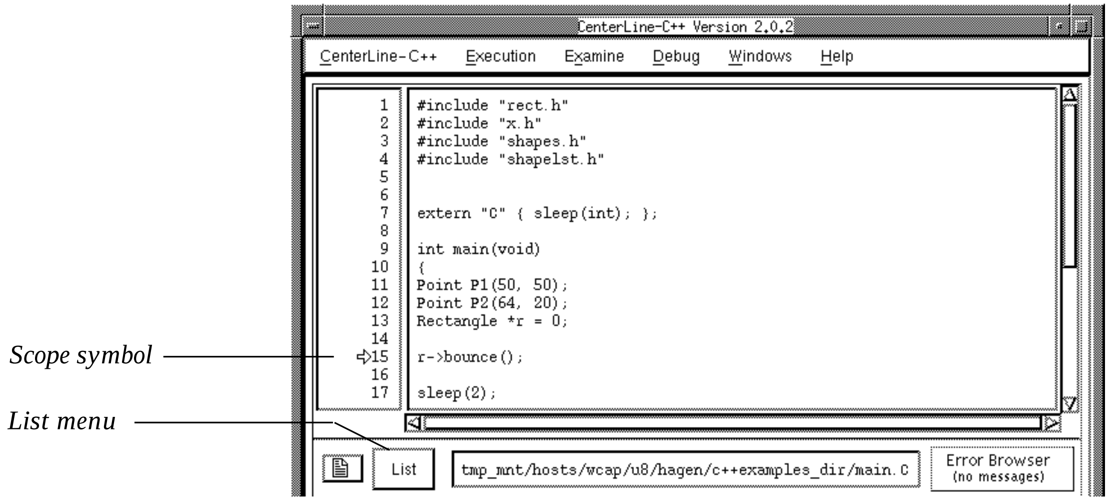


Setting breakpoints

To examine how the Bounce program executes before the segmentation fault occurs, you can move up the execution stack to see where the **DrawableShape::bounce()** routine is called:

In the Button panel, click on the **Up** button.

The Source area now displays **main.C** and uses the Scope symbol to show the current location in the call stack.



Whenever you list a new file, the debugger adds it to the List menu, which makes it convenient to navigate among your source files.

To examine data structures at the current location in the call stack, you can set a breakpoint in **main.C** before the program calls the **bounce()** function. To set the breakpoint:

In the Source area, click on line number 15 in the left margin.

The Breakpoint symbol appears next to the line number, and the Workspace indicates that debugging item 1 has been set.

To run the Bounce program:

In the Button panel, click on the **Run** button.

The Run Window opens, and the Bounce program executes until it reaches the breakpoint. In the Main Window the Execution symbol moves to line number 15, and a break level is generated in the Workspace.

```
pdm (break 1) 4 ->
```

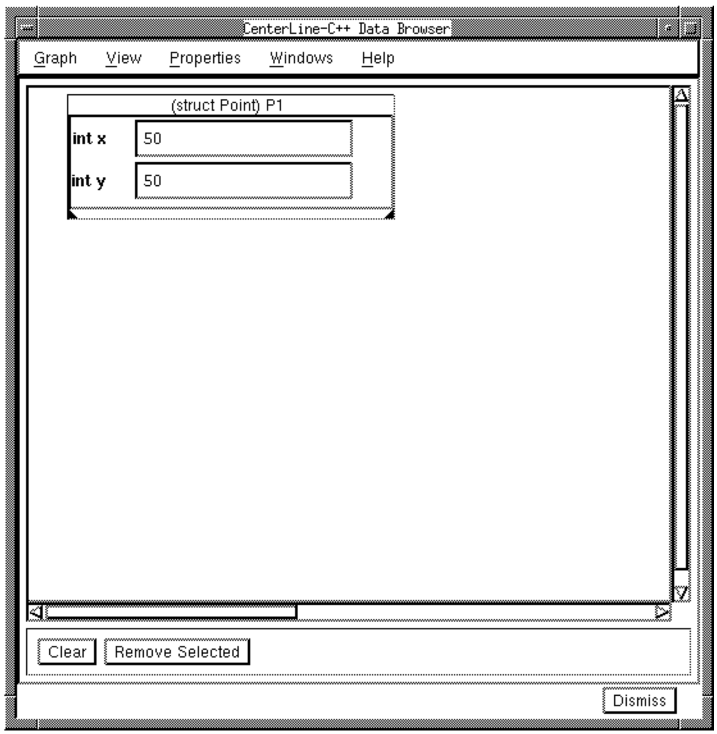
Finding the error with the Data Browser

While at a break level, you can use the Data Browser to examine the state of any data structure that is currently in scope. To begin, examine **P1**:

1 Select the text string **P1** on line 11.

2 Display the **Examine** menu and select **Display<Selection>**.

The Data Browser opens and displays a graphical data item for the point object called **P1**. The Workspace indicates that debugging item 2 is set. The breakpoint is debugging item 1, and the "display" is debugging item 2.



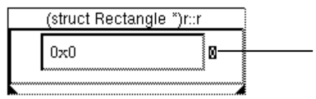
Since **P1** looks correct, now examine the rectangle ***r**. To do so, position the Main Window so it does not overlap the Data Browser. Then, in the Main Window, type following in the Workspace:

```
pdm (break 1) 4 -> display *r
Cannot access memory at address 0x0.
Disabling display 3 to avoid infinite recursion.
```

The Data Browser does not display the data item for ***r**. Since the rectangle cannot be displayed, examine the pointer to it:

1. Select the text string **r** on line 13.
2. From the Data Browser, display the **Graph** menu and select **New Expression <Selection>**.

The data item for **r** contains a pointer box with an X through it, which indicates **r** is a null pointer. Thus, line 15 dereferences a null pointer, which causes the segmentation fault.



Null pointer symbol

To dismiss the Data Browser:

Select the **Dismiss** button.

Fixing the error and running the program

To fix the error, invoke your editor on **main.C**:

1. In the left margin of the Source area, move the mouse pointer over number 15.
2. Press the Right mouse button to display the pop-up menu and select **Edit line 15**.

Your editor opens **main.C** and positions the cursor at line 15. If you examine the declaration for ***r**, notice that it has been initialized to 0, but the rectangle has not been created before it is bounced. To create the rectangle, do the following in your editor:

1. Insert the following line before line 15 (line 15 contains **r->bounce();**):

   ```
   r = new Rectangle(P1, P2);
   ```

2. Save the file and quit from the editor.

To rebuild and reload the Bounce program:

```
pdm (break 1) 5 -> make tutorial
...
pdm (break 1) 6 -> debug tutorial
debug: Deleting all debugging items.
Debugging program ‘tutorial’ (previous program ‘tutorial’)
```

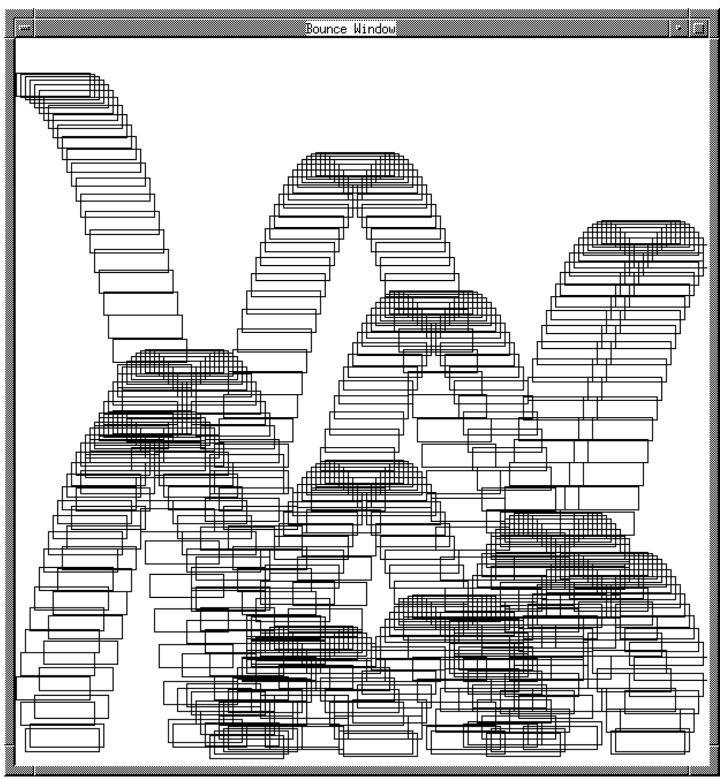
To verify that your fix is correct:

1 In the Main Window, set a breakpoint on the line:

```
sleep(2);
```

2 Click on the **Run** button.

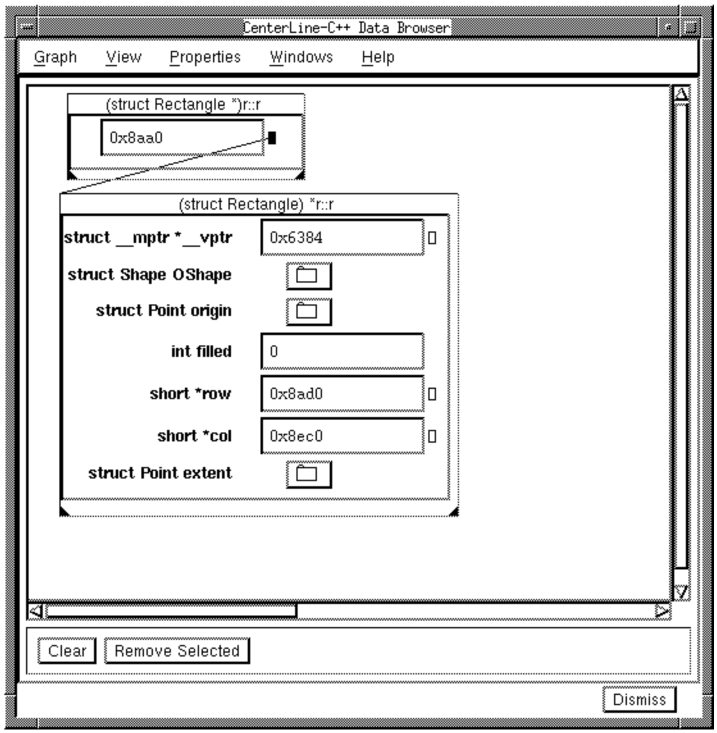
The Bounce window appears and a rectangle bounces in it.



To examine the pointer again:

```
pdm (break 1) 7 -> display r
display (6) set on expression'r::r'.
pdm (break 1) 8 -> display *r
display (7) set on expression'*r::r'.
```

The Data Browser displays the data item for the Rectangle, as shown in the following illustration:



To continue execution of the program:

> In the Main Window's Button panel, click on **Continue**.

The Bounce program completes successfully. You have now completed the tutorial.

6 Debugging with CenterLine-C++

This chapter contains tasks that you can perform with the CenterLine-C++ debugger, pdm.

Some of the tasks refer to the CenterLine-C++ online code examples. If you have not yet set up the CenterLine-C++ examples directory, refer to *"Setting up the examples directory" on page 9* *for instructions.*

Task: Selecting an editor to use with the debugger

Once you select an editor to use with the debugger, you use it for the duration of your session. You cannot change editors within a debugging session.

The vi editor

If you use the **vi** text editor, you're all set. By default, the debugger uses **vi**.

The GNU emacs editor

If you use the GNU **emacs** editor, set the EDITOR environment variable before you start the debugger:

```
% setenv EDITOR emacs
% centerline-c++
```

NOTE The debugger supports only GNU **emacs**, and the version of GNU **emacs** must be capable of running as an X Window System client.

See also

"Editing source code" on page 145

edit on page 219

Task: Starting up the debugger

You start up the debugger by invoking the **centerline-c++** command at the shell:

centerline-c++ [*switches*] [*a.out* [*core* | *process_id*]]

The **centerline-c++** command is installed in a **CenterLine/bin** directory, which could be installed anywhere on your system. If **CenterLine/bin** is not in your path or if you need to know the absolute path for **CenterLine/bin**, see your system administrator.

When the debugger starts, it displays a welcome dialog and then the Main Window.

Specifying files and processes to be debugged

Use one of the following arguments with the **centerline-c++** command:

executable	Loads a fully-linked executable for debugging.
executable core	Loads a fully-linked executable and a core file generated by the executable.
executable process_id	Loads a fully-linked executable and attaches to a process that is running the executable.

You can also load a file for debugging after you start the debugger. See "Loading files for debugging" on page 142 for details.

Specifying the user interface

Use one of the following switches with the **centerline-c++** command:

-motif	Motif GUI, which is the default on all platforms except for Sun.
-openlook	OPEN LOOK GUI, which is the default on Sun systems.
-ascii	Nongraphical user interface that has a single work area, the Workspace.

If you prefer to use a user interface that differs from the default for your platform, you can set an X Window System resource to do so. By setting an X resource, you don't need to specify the **-motif** or **-openlook** command-line switch with **centerline-c++**. For example, to choose Motif as the default user interface, add the following resource to your X resources file:

```
CenterLine-C++*Model: Motif
```

To choose OPEN LOOK as the default, add one of the following resources to your X resources file:

```
CenterLine-C++*Model: openlook
CenterLine-C++*Model: openlook2d
CenterLine-C++*Model: openlook3d
```

NOTE Before running the Motif or OPEN LOOK version of the debugger, make sure to set your **DISPLAY** environment variable to the display on which you want to use the debugger. Otherwise, the debugger may display on another machine or exit with an error.

Startup file

You can customize the debugger at startup with the **.pdminit** file. See "Customizing your startup file" on page 191 for more information.

See also

centerline-c++ in the Man Browser

debug on page 212

"Loading files for debugging" on page 142

"Customizing your startup file" on page 191

Task: Finding out more about the debugger

To find out more about the debugger, you can use the **help** command and Man Browser. If you run into problems, you can also send email to CenterLine with the **email** command.

Using the help command

For a brief description of any Workspace command, use the **help** command. For example:

```
pdm -> help debug
debug - Print the name and args of the program being debugged
debug prog [core | pid] - Begin debugging <prog>
                          Access core file if specified, otherwise attach
                          to the running process specified by pid

pdm -> help email
email - send e-mail to CenterLine Software
email file - send file to CenterLine Software
```

Using the Man Browser

The Man Browser provides detailed information about Workspace commands, shell commands, and other topics (such as X resources). You can display the Man Browser using two different methods:

- From any primary window in the debugger, display the **Windows** menu and select **Man Browser**.
- In the Workspace, use the **man** command and specify the Workspace command as an argument:

  ```
  pdm -> man command
  ```

NOTE The **man** command can handle only Workspace commands as arguments. To display the Man Browser entries for shell commands or conceptual topics, open the Man Browser from the **Windows** menu and use the **Categories** and **Topics** lists, as shown in the illustration on page 124.

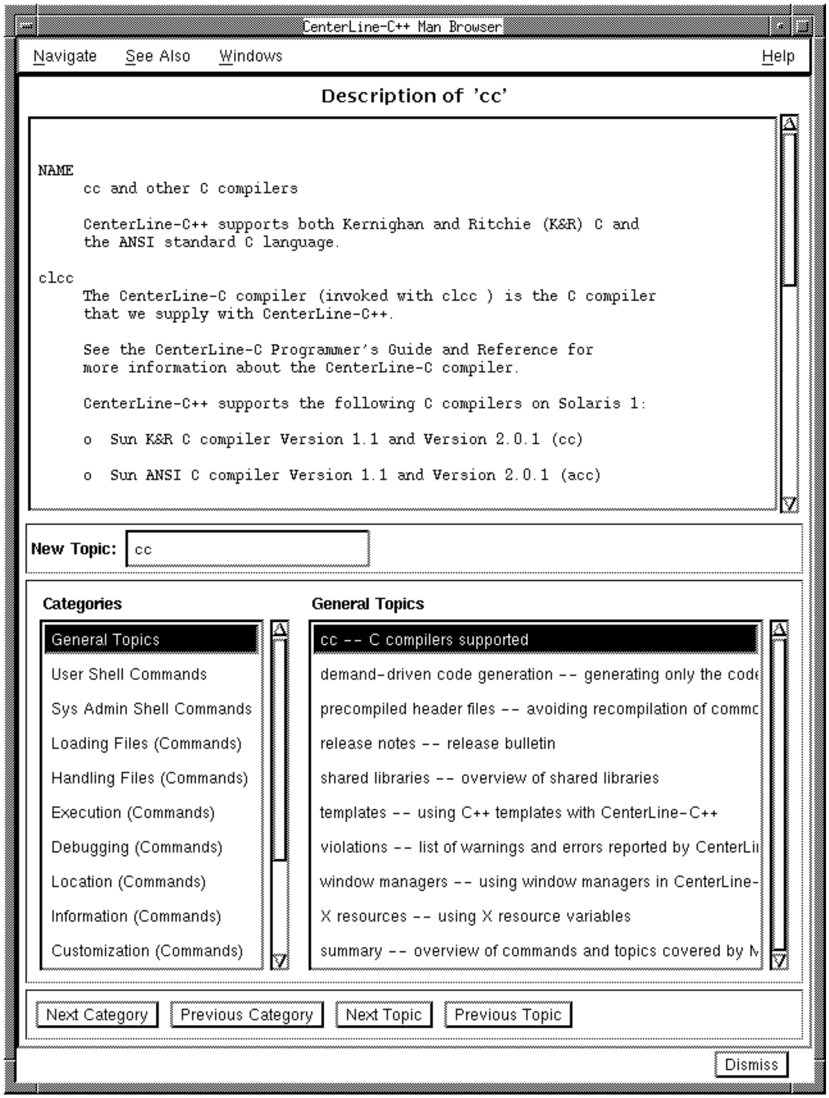
Once you display the Man Browser, you can select different topics in it and find related topics using the following methods:

- To select a different topic in the Man Browser, select an item in the **Categories** list and an item in the **Topics** list. You can also type the topic name directly into the **New Topic** text field.

- To find a topic that is related to the currently displayed topic, display the **See Also** menu in the menu bar and select an item from the menu.

  There is also a pop-up **See Also** menu. To display it, move the mouse cursor over the displayed topic and press the Right mouse button.

The following illustration shows the Man Browser.



Although the Man Browser doesn't provide a facility for printing topics, you can print the topics using a shell. To do so go to the **docs** directory and use the print command available with your operating system (such as **lpr**):

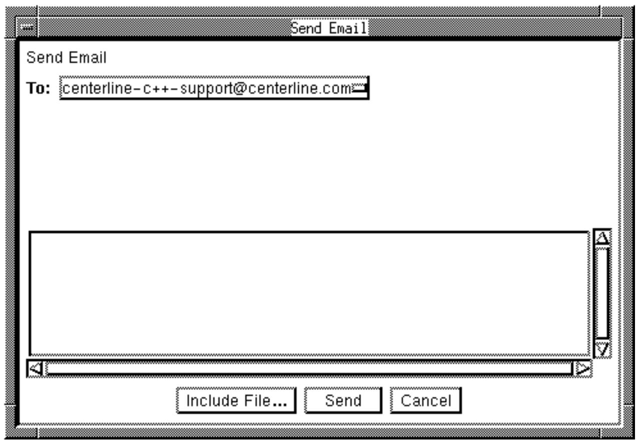
```
% cd CenterLine/clc++/docs
% lpr filename
```

Sending email to CenterLine

If you encounter a problem or have a suggestion, you can send email to CenterLine without leaving the debugger. You can send email using two different methods:

- In the Main Window, display the **CenterLine-C++** menu and select **Send Email**.
- In the Workspace, enter the **email** command.

In the **Send Email** dialog, enter the text of your message in the text box and if applicable, select the **Include File** button to include a sample program or session log. When you are satisfied with the message, select the **Send** button.



See also

email on page 220

help on page 224

man on page 232

Task: Using menus and text fields

Using the PRIMARY text selection as input to a command

Any menu command that has **<Selection>** as part of its name, such as **Display <Selection>**, uses the PRIMARY text selection as its input argument without prompting you for confirmation. The **PRIMARY text selection** is the selection created when you select a range of text by dragging with the Left mouse button.

Before you choose a menu command, be aware of any text that you may have selected in the debugger or other X application windows. The debugger uses this text as input to the command unless you explicitly select text to be used instead.

If desired, you can customize the debugger so that it displays the PRIMARY text selection in a dialog box for you to confirm. Once you select the **OK** button in the dialog, the debugger uses the text as input to the menu command. To do so:

1 Add the following line to your **.Xdefaults** file.

```
CenterLine-C++*ConfirmSelnUse: True
```

2 Load your **.Xdefaults** file into the X resource database with the **xrdb** command.

```
% xrdb ~/.Xdefaults
```

3 Start the debugger with the **centerline-c++** command, as described in "Starting up the debugger" on page 120.

Using pop-up menus

The debugger provides pop-up menus in most of its windows.

Table 7 Pop-up Menus

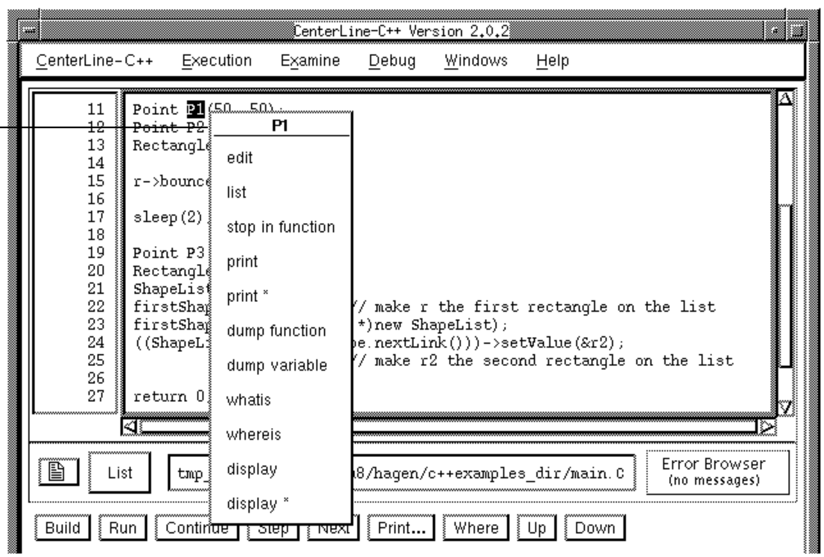
Location	Menu	How to Display It
Source code in the Source area of the Main Window	File Options menu	Press the Right mouse button.
Line numbers in the Source area of the Main Window	Line Number Options menu	Move the mouse pointer over a line number and press the Right mouse button.

Table 7 Pop-up Menus

Location	Menu	How to Display It
Workspace in the Main Window	Workspace Options menu	Press the Right mouse button.
Source area or Workspace in Main Window	Expression Options menu	Select an expression or identifier or move the mouse pointer over an expression or identifier. Then, hold down Shift and press the Right mouse button.
Title of data item in Data Browser	Data Item menu	Move the mouse pointer over the title of a data item and press the Right mouse button.
Reference entry in Man Browser	See Also menu	Move the mouse pointer over the reference entry and press the Right mouse button.

The following illustration shows the Expression Options menu.

Press Shift plus Right mouse button to display the Expression Options menu



Using shortcuts for Expression Options menu

You can use the following shortcuts instead of selecting the **whatis** or **print** menu commands from the **Expression Options** menu. First, select an expression or identifier, or move the mouse pointer over an expression or identifier, making sure nothing else is selected.

- To issue the **print** command, hold down the Shift key and press the Left mouse button.
- To issue the **whatis** command, hold down the Shift key and press the Middle mouse button.

Editing text fields in dialogs and windows

All text fields in the debugger support the subset of **emacs** keyboard shortcuts shown in Table 8.

Table 8 Emacs Keyboard Shortcuts in Text Fields

Shortcut	Meaning
Control-a	Move to the beginning of the line.
Control-e	Move to the end of the line.
Control-b	Move backward one character.
Control-f	Move forward one character.
Meta-b	Move backward one word.
Meta-f	Move forward one word.
Control-n	Move to the next line.
Control-p	Move to the previous line.
Control-d	Delete the next character.
Control-u	Delete to the beginning of the line.
Control-k	Delete to the end of the line.
Control-w	Delete the previous word.

To change these shortcuts, refer to **X resources** in the Man Browser.

See also

"Copying and pasting text between windows" on page 129

Task: Copying and pasting text between windows

You can copy and paste between any debugger window in Motif and other Motif applications and between any debugger window in OPEN LOOK and other OPEN LOOK applications. To do so, use the standard copy and paste methods for Motif and OPEN LOOK.

Since Motif applications (and X applications that use the MIT Athena widget set) use the PRIMARY selection and OPEN LOOK applications use the CLIPBOARD selection, copying and pasting between dissimilar GUIs can require additional steps.

NOTE This task applies only to users of the OPEN LOOK version of the debugger.

Copying the PRIMARY selection from OPEN LOOK

To copy and paste the PRIMARY selection from any window in the OPEN LOOK debugger to an **xterm**:

1 Select the text from the debugger. This copies the text to the PRIMARY selection.

2 Move the mouse pointer into the **xterm** window and press the Middle mouse button to paste the PRIMARY selection.

Using the CLIPBOARD selection with xterm

By default, **xterm** uses the PRIMARY selection rather than the CLIPBOARD selection. To customize **xterm** so it can use either type of selection:

1 Define the Copy and Paste keys by adding the following lines to your **.Xdefaults** file:

```
! copy and paste from an xterm to/from CLIPBOARD
XTerm*vt100.translations: #override \n\
<Key>L6: start-extend() select-end(CLIPBOARD) \n\
<Key>L8: insert-selection(CLIPBOARD) \n
```

This example uses the standard key bindings for Copy and Paste (the L6 and L8 keys on a Sun keyboard), however you can substitute different key bindings if desired.

2 Load these definitions into your X resource database with the UNIX **xrdb** command:

```
% xrdb ~/.Xdefaults
```

To copy text from an **xterm** to the OPEN LOOK debugger, or from the OPEN LOOK debugger to an **xterm**:

1 Select the text from the first window, which copies the text to the PRIMARY selection.

2 Press the Copy key, which copies the PRIMARY selection to the CLIPBOARD selection.

3 Move the mouse pointer into the second window and press the Paste key to paste the CLIPBOARD selection.

Pasting the PRIMARY selection in OPEN LOOK

By default, the OPEN LOOK version of the debugger uses the CLIPBOARD selection when pasting. If you try to paste a nonexistent CLIPBOARD selection, however, the debugger pastes the PRIMARY selection. To customize the debugger so it can paste either type of selection:

1 Add the following lines to your **.Xdefaults** file:

```
! copy and paste from PRIMARY selection as in
! many X applications
*OI*OI_multi_text.Translations:#override\n\
Shift <Key>L8:insert_selection(PRIMARY)\n
*OI*OI_entry_field.Translations:#override\n\
Shift <Key>L8:insert_selection(PRIMARY)\n
```

This example uses the Shift key and the standard key binding for Paste (L8 key), however, you can substitute different key bindings if desired.

2 Load these definitions into your X resource database with the UNIX **xrdb** command:

```
% xrdb ~/.Xdefaults
```

3 To paste the PRIMARY selection into the debugger, press Shift and the Paste key.

Task: Invoking Workspace commands

You can enter commands in the Workspace at the debugger's prompt:

```
pdm ->
```

The **pdm** prompt stands for process debugging mode. Although the CenterLine-C++ debugger supports only this single mode of debugging, other CenterLine products support multiple modes. The **pdm** prompt is shared by all CenterLine products that support process debugging mode.

Workspace commands take the following form:

command_name [*switches*] [*arguments*]

To cancel a Workspace command, press Control-c.

Getting help on commands

For a complete list of all the commands that you can enter in the Workspace, enter the **help** command with no arguments. Table 9 outlines the Workspace commands that you use for key debugging tasks.

For information on specific commands, you can:

- Refer to Chapter 7, "Command Reference," on page 201.
- Open the Man Browser. To do so, in any primary window, display the **Windows** menu and select **Man Browser**.
- Use the **help** command with the command name as an argument to display a brief description of the command.

  pdm -> **help** *command*

- Use the **man** command with the command name as an argument to display detailed information about the command.

  pdm -> **man** *command*

Table 9 Workspace commands by function

Function	Commands
Loading files	**build**, **debug**, **make**, **source**, **attach**, **detach**
Handling files	**list**, **edit**, **use**, **cd**, **file**
Execution	**run**, **rerun**, **cont**, **step**, **next**, **stepout**, **reset**
Debugging	**delete**, **status**, **stop**, **when**
Location	**up**, **down**, **where**, **whereami**
Information	**contents**, **help**, **man**, **print**, **whatis**, **whereis**, **dump**, **display**
Customization	**alias**, **unalias**
Signals	**catch**, **ignore**
Session	**quit**, **setenv**, **unsetenv**, **printenv**, **gdb**, **gdb_mode**
Miscellaneous	**sh**, **shell**, **email**, **assign**, **set**
Machine level debugging	**nexti**, **stepi**, **stopi**, **listi**

Invoking commands from a file

You can use the **source** command to execute Workspace commands from a file:

```
pdm -> source file
```

Using gdb commands

The GNU Debugger, **gdb**, is a popular source-level debugger available from the Free Software Foundation. The debugger provides access to **gdb**, but CenterLine does not provide support for this feature.

You can enter individual **gdb** commands with the **gdb** command:

```
pdm -> gdb command
```

You can also invoke **gdb** in the Workspace with the **gdb_mode** command:

```
pdm -> gdb_mode
(gdb)
```

Enter the **pdm** command to return to the debugger. The debugger does not restart a new session; it continues numbering history entries. When you display your command history (as described in “Displaying and manipulating your input history” on page 137), any **gdb** commands you entered in **gdb** mode are also included in the list.

For more information about **gdb**, use the **help** command while in **gdb** mode or refer to the documentation that is available using anonymous **ftp**, as described in “Distribution” on page iii.

Examples

To try out the following examples, you need to set up and load the **bounce** program.

NOTE If you have not yet set up the CenterLine-C++ examples directory, refer to “Setting up the examples directory” on page 9 for instructions.

To begin, use the **cd** command to change to the examples directory:

```
pdm -> cd ~/c++examples_dir
```

Use the **make** command to build the **bounce** program.

```
pdm -> make bounce
```

To load the bounce executable into the debugger, use the **debug** command:

```
pdm -> debug bounce
```

You can use the **whatis** command to display the use of a name. To see the declaration for the function **DrawableShape::bounce()**, you can type:

```
pdm -> whatis DrawableShape::bounce
char DrawableShape::bounce(void);
```

To list the file **xfixed.C**, in the Source area, you can type:

```
pdm -> list xfixed.C
Listing file 'c++examples_dir/xfixed.C', line 1
```

As you can see from browsing in the Source area, the **xfixed.C** file provides the interface between the **bounce** program and the X Window System.

See also

"Using aliases for Workspace commands" on page 136

"Saving your debugging session" on page 190

"Customizing your startup file" on page 191

gdb on page 222

gdb_mode on page 223

source on page 247

Task: Invoking shell commands

The debugger provides two ways to invoke shell commands:

- Passing commands to a subshell
- Invoking a shell

Passing commands to a subshell

Use the **sh** command to execute a command in a *Bourne* subshell (**/bin/sh**), where *argument* is the command to be passed to the shell:

```
pdm -> sh argument...
```

Use the **shell** command to execute a command in your *default* subshell, which is specified by your SHELL environment variable:

```
pdm -> shell argument...
```

Invoking your default shell

Use the **sh** or **shell** commands without arguments to invoke the Bourne Shell or your default shell (respectively):

```
pdm -> sh
#
pdm -> shell
baxter 1 %
```

To exit from the shell, use the **exit** shell command. You can also use the **pdm** shell command, but it starts a new session, numbering history entries from 1.

Example

Although you can search for strings in the Source area (use **Find** from the pop-up menu in the Source area), you might prefer to use the **grep** command to search a large number of files:

```
pdm 2 -> cd ~/c++examples_dir
pdm 3 -> sh grep 'bounce' *.[cC]
main.C:r->bounce();
mainfixed.C:r->bounce();
shapes.C:void DrawableShape::bounce()
x.C:     * Open the basic bounce window *
x.C:     *    Unmap and close the bounce window *
xfixed.C:     * Open the basic bounce window *
xfixed.C:     *   Unmap and close the bounce window *
```

Task: Using aliases for Workspace commands

Using aliases

At startup, the debugger automatically creates aliases for the shell commands **pwd** and **ls**. To see all the aliases currently defined, use the **alias** command with no arguments:

```
pdm -> alias
ls          sh ls
pwd         sh pwd
n           next
s           step
```

Creating aliases

To create an alias, use the **alias** command, where *alias_name* is the name of the alias and *command* is the command to be executed:

```
pdm -> alias alias_name command
```

To save aliases across sessions, see "Customizing your startup file" on page 191.

Examples

To create an alias that sends a file to the printer, and then print the file **~/c++examples_dir/main.C**:

```
pdm 25 -> alias lpr sh lpr
pdm 26 -> lpr ~/c++examples_dir/main.C
```

The following aliases provide keyboard shortcuts for common Workspace commands:

```
alias c cont
alias p print
alias d delete
alias l list
alias st status
alias w where
alias q quit
```

See also

alias on page 203

unalias on page 255

Task: Editing Workspace input

The Workspace supports a range of input features, such as command history, name completion, and in-line editing. These features are based on similar ones found in the **tcsh** shell (an extended version of the **csh** shell) and the **emacs** editor.

The Workspace Options menu

The Workspace provides a pop-up menu, the **Workspace Options** menu, with commands for editing, clearing, and saving the Workspace. To display the menu, move the mouse pointer in the Workspace and press the Right mouse button.

Evaluating variables and expressions

You can evaluate the value of variables and expressions (providing they are in scope) in the Workspace. For example:

```
pdm (break 1) 42 -> P1 = P2
(struct Point) = {
  int x = 64;
  int y = 20;
}
```

Displaying and manipulating your input history

Use the debugger's **history** command to display previous input:

```
pdm -> history
```

The Workspace features a history mechanism modeled after the **csh** and **tcsh** shells. As in the **csh** shell, you can execute previous lines of input using a history character followed by an argument. The debugger's history character is **#**, while the **csh** shell's history character is **!**. Table 10 summarizes the syntax for repeating previous input.

Table 10 Syntax for Repeating Previous Input

Task	Syntax
Display the previous input line in the history list	<Control-p>
Display the next input line in the history list	<Control-n>
Display, searching backward in the history list, the next input line that begins with *text*	*text*<Control-p>

Table 10 Syntax for Repeating Previous Input (Continued)

Task	**Syntax**
Display, searching forward in the history list, the next input line that begins with *text*	*text*<Control-p>
Repeat most recent line of input	##<Return>
Display most recent line of input so you can edit it	##<Space>
Repeat a particular line of input	
By matching the text at the beginning of the input line	#*text*
By matching the input line number	#*n*
By specifying the *n*th previous input line	#-*n*
Expand the last token of the previous input line	#$<Space>
Expand all but the first token of the previous input line	#*<Space>
Expand the *n*th token on the previous input line, where tokens are numbered beginning at 0	#:*n*<Space>

Line editing

The Workspace supports line editing of input similar to that available in the **emacs** editor. Table 11 outlines the line editing commands available.

Table 11 Line Editing Commands

Key Sequence	**Action**
Control-a	Moves the cursor to the beginning of the line.
Control-e	Moves the cursor to the end of the line.
Control-f	Moves the cursor forward one character.
Control-b	Moves the cursor backward one character.
Esc-f	Moves the cursor forward one word.
Esc-b	Moves the cursor backward one word.
Control-d	Deletes the next character.
Control-h	Deletes the previous character.
Control-k	Deletes characters from the cursor until the end of the line.

Table 11 Line Editing Commands (Continued)

Key Sequence	Action
Control-u	Deletes the line.
Esc-d	Deletes the next word.
Esc-Delete	Deletes from the cursor to the beginning of the previous word.
Control-p	Displays the previous input line in the history list.
Control-n	Displays the next input line in the history list.
Esc-<	Displays the first input line in the history list.
Control-t	Transposes the previous two characters.
Esc-t	Transposes the words on either side of the cursor.
Control-v *char*	Inserts the key sequence *char* as text without interpreting key definition. You can use this to insert control characters.
Control-y	Pastes the text that was previously deleted.
Control-j	Executes this line.
Control-l	Skips a line.
Esc-u	Changes the next word to uppercase.
Esc-l	Changes the next word to lowercase.
Esc-c	Capitalizes the next word.
Control-_	Undoes the last edit.

Using name completion

The Workspace provides name completion for all commands.

In addition, for commands that take a program symbol (such as a variable, function, or object) as an argument, the Workspace provides name completion for these symbols. This feature can be useful if you suspect there are overloaded functions in the code that you are debugging. You complete commands by entering some text and pressing the Tab key twice:

```
pdm -> text<Tab><Tab>
```

You complete symbols by entering the command (only commands that takes a program symbol as an argument), entering some of the symbol name (text), and pressing the Tab key twice:

```
pdm -> command text<Tab><Tab>
```

If the completion is ambiguous, the unambiguous portion is completed and all possible matches are listed.

Redirecting output

Just as you can redirect output of commands at the shell, you can redirect the output of a subset of the Workspace commands with the following symbols:

#> *file*	Redirects the command output to *file*. Overwrites *file* if it exists.
#>> *file*	Appends the command output to the contents of *file*.

Note that the debugger's redirection symbols start with #.

NOTE You cannot redirect output for the following commands: **run**, **step**, **next**, **cont**, **reset.**

Examples

This section provides a variety of examples of editing Workspace input.

The following example shows how to use Control-p and **##** to repeat previous Workspace input:

```
pdm 4 -> cd ~/c++examples_dir
pdm 5 -> make bounce
pdm 6 -> debug bounce
pdm 7 -> <Control-p><Control-p>   expands to...
pdm 7 -> make bounce
pdm 7 -> make bounce
pdm 8 -> ##<Space>                expands to...
pdm 8 -> make bounce
```

The next example builds on the previous one and shows how to match text or line numbers as a method for repeating previous Workspace input:

```
pdm 9 -> #5<Space>                expands to...
pdm 9 -> make bounce
pdm 10 -> #-4<Space>              expands to...
pdm 10 -> debug bounce
pdm 11 -> #de<Space>              expands to...
pdm 11 -> debug bounce
```

To expand tokens from the previous line of input:

```
pdm 10 -> make static
pdm 11 -> debug #*<Space>       expands to...
pdm 11 -> debug static
pdm 12 -> make static bounce bounce_fixed
pdm 13 -> debug #:2<Space>      expands to...
pdm 13 -> debug bounce
```

To list the C source files in your current working directory using Shell metacharacter expansion, issue the following Workspace command:

```
pdm -> ls *.c
Vector.c      table.c
```

To redirect the output of the **printenv** command to the file **my_vars**:

```
pdm -> printenv #> my_vars
```

To use command and name completion to list the source file containing the **doDraw()** routine:

```
pdm -> lis<Tab><Tab>       completes to...
pdm -> list do<Tab>Tab>    ambiguous, completes to...
doDraw__13DrawableShapeFv     do_ipfx__7istreamFi
    do_opfx__7ostreamFv
pdm -> list doD<Tab>Tab>    completes to...
pdm -> list doDraw__13DrawableShapeFv
Listing file ‘c++examples_dir/shapes.C’, line 89
```

See also

"Invoking Workspace commands" on page 131

"Saving your debugging session" on page 190

Task: Loading files for debugging

You can load the following files for debugging

- A fully-linked executable
- A fully-linked executable and a corefile generated by it
- A fully-linked executable and a process running that executable

Each time you debug an executable, the debugger unloads the previously loaded executable.

Loading an executable

Enter the following command, where *executable* is the name of the fully-linked executable:

```
pdm -> debug executable
```

Loading an executable with a corefile

Enter the following command, where *executable* is the name of the fully-linked executable and *core* is the name of its associated corefile:

```
pdm -> debug executable core
```

Loading an executable with a running process

Enter the following command, where *executable* is the name of the fully-linked executable and *process_id* is the process ID of the process running the executable:

```
pdm -> debug executable process_id
```

When you load an executable with a process ID, the debugger attaches to the process. To detach from the process, use the **detach** Workspace command.

See also

"Debugging an executable with a corefile" on page 178

"Debugging a running process" on page 182

debug on page 212

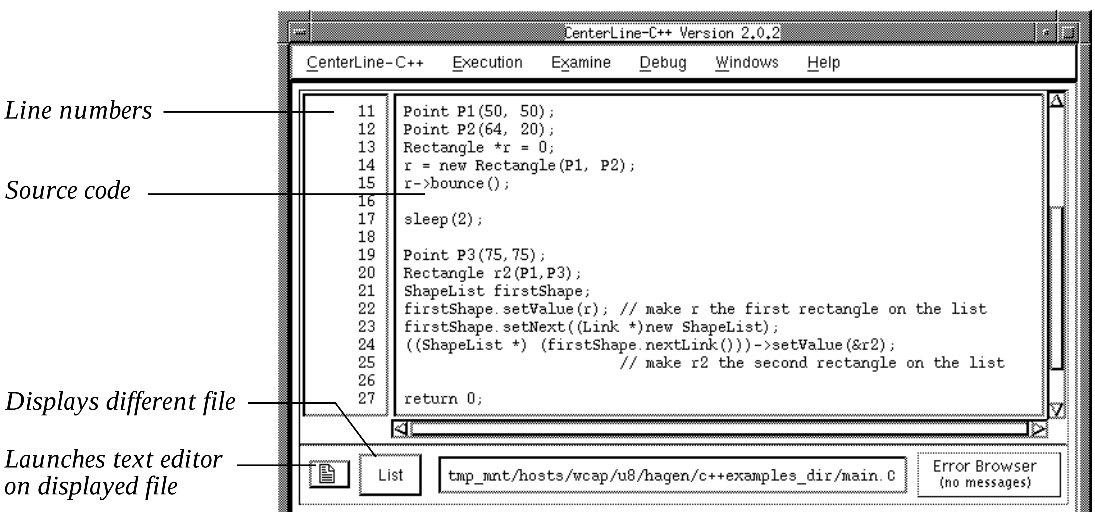
Task: Listing source code

Displaying source code in the Source area

You can list source code in the Source area of the Main Window in a number of different ways:

Main Window	Select the filename (as text) from a debugger window or another X application window. Then, from the **Examine** menu, select **List <Selection>**.
	From the **List** menu below the Source area, select a previously listed file or select **New File**, which opens a file selection dialog box.
	Select the function (as text) or move the mouse pointer over the function. Display the **Expressions Options** menu by pressing the Shift key and the Right mouse button, and select the **list** command.
	Use the **list** Workspace command.
Error Browser	Select a warning or error message.

The following illustration shows the Source area.



You can also use the **listi** Workspace command to display machine instructions in the Workspace.

Displaying the source files for current executable

To display all the known source files for the current executable that is loaded in the debugger, use the **contents** command with the **all** argument:

```
pdm -> contents all
```

The **contents** command lists only the files that were compiled with debugging information (the **-g** switch).

Displaying the functions defined in a source file

You can also display the objects or functions that are declared or defined in a particular source file. To do so, use the **contents** command and specify the filename as an argument:

```
pdm -> contents filename
```

Occasionally, the **contents** *filename* command displays only a partial list of the objects declared or defined in *filename*.

Example

The following example uses the **contents** command with the Bounce program:

```
pdm -> cd ~/c++examples_dir
pdm -> make bounce
pdm -> debug bounce
pdm 25 -> contents xfixed.C
Contents of source: xfixed.C
char __std__xfixed_C_openWindow_();
char __sti__xfixed_C_openWindow_();
char closeWindow(void);
char drawCenterLine(void);
char drawCircle(short, short, short, int);
char drawRect(int, int, int, int, int);
char makeFilled(void);
char openWindow(int, int, int, int);
```

See also

contents on page 211

list on page 228

listi on page 230

Task: Editing source code

You can edit source code by:

- Invoking your editor from the debugger
- Attaching an existing editing session to the debugger (GNU **emacs** only)

Invoking your editor from the debugger

You can invoke your editor in a variety of different ways:

Main Window

Select the filename (as text) from a debugger window or another X application window. Then, from the **Examine** menu, select **Edit <Selection>**.

Select the Edit symbol below the Source area.

From the Source area pop-up menu, select **Edit**.

From the **Line Number Options** pop-up menu at the left of the Source area, select **Edit**.

Select the filename (as text) or move the mouse pointer over the filename. Display the **Expressions Options** menu by pressing the Shift key and the Right mouse button and select the **edit** command.

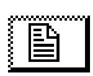
Use the **edit** Workspace command.

Error Browser

Select the Edit symbol at the left of a warning or error message.

The following illustration shows the Edit symbol, which provides a convenient way to edit files from the Main Window and Error Browser:

Edit symbol



When you quit from the debugger, it removes any text editor windows along with all debugger windows.

If you use the **emacs** editor and lower or iconify the window between edits, the debugger cannot always raise or deiconify the **emacs** window.

Connecting GNU Emacs to the debugger

To connect an existing GNU Emacs session to the debugger:

1 Put the following lines of ELISP code in your **.emacs** startup file. In this example, replace *directories* with the absolute path to your CenterLine directory and *arch-os* with your platform-specific directory, such as **sparc-sunos4** or **sparc-solaris2**.

```
(setq load-path ( cons "directories/CenterLine/lib/lisp" load-path))
(setq exec-path ( cons "directories/CenterLine/arch-os/bin" exec-path))
(require ’clipc)
```

The **exec-path** line tells Emacs which directories to search when executing a binary. The **emacs** edit server uses the **clms_query -b** command to get the correct session ID, and if it cannot locate the **clms_query** binary, the connection fails.

2 Load the ELISP lines in your current Emacs session. To do so, select these lines of ELISP and evaluate the region in Emacs (**M-x eval-region**).

3 Establish a connection to the debugger by using the Emacs command **M-x cl-edit**.

The next time you invoke Emacs, the ELISP lines are automatically loaded from your **.emacs** startup file. You just need to establish a connection with **M-x cl-edit**.

When you quit from the debugger, any Emacs windows that you connected to it remain open.

See also

edit on page 219

"Selecting an editor to use with the debugger" on page 119

Task: Building and reloading executables

Using make in the Workspace

For your convenience, you can use the **make** command in the Workspace to rebuild your fully linked executable. The **make** command behaves the same way in the Workspace as it does in a shell.

To build your default target:

```
pdm -> make
```

To build a specific target:

```
pdm -> make target...
```

You can also use other **make** options. For a complete list of options, refer to the **make** manual page that was supplied with your operating system.

NOTE The directory containing the **make** command, which varies from platform to platform, must be in your PATH environment variable. For example, on the **sparc-solaris2** platform, **make** is installed in **/usr/ccs/bin**.

Loading rebuilt executables

To load rebuilt executables into the debugger, you can either:

- Select the **Build** button in the Button panel of the Main Window
- Use the **build** command:

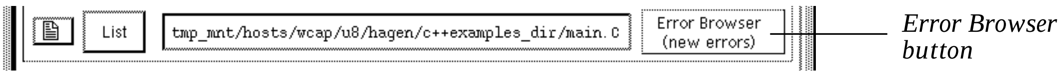
```
pdm -> build
```

If the executable has not changed since it was last loaded, **pdm** does not reload it.

Task: Finding and fixing errors

Two types of errors can occur when you use the debugger:

- As you compile and build your program, the Error Browser button in the Main Window indicates when compiler and **make** errors occur. You can fix these kinds of problems immediately using the Error Browser.



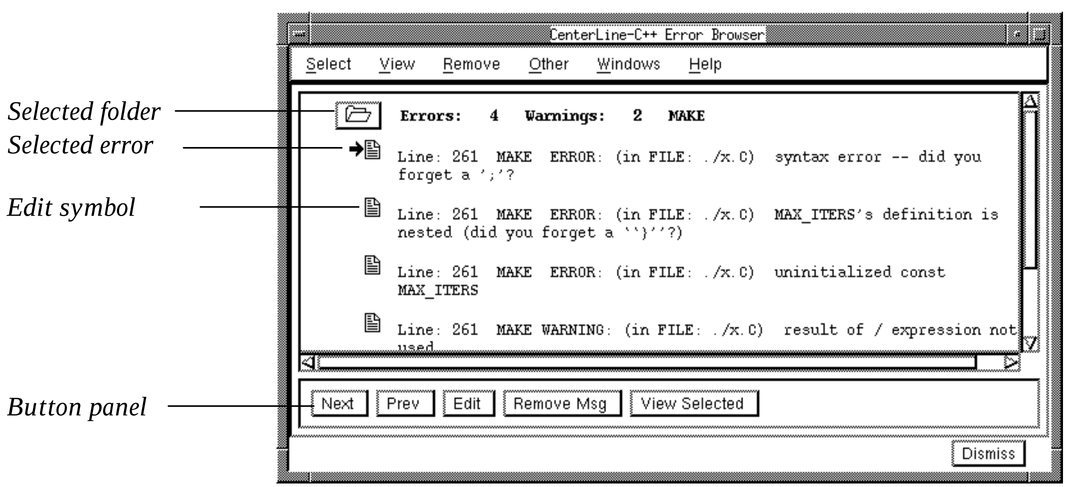
- As you execute and debug your program, the Workspace indicates when errors or signals occur. The Workspace creates a break level when your program generates a signal. You can work at the break level to learn more about the problem before fixing it.

```
pdm 3 -> run
Program received signal 10, Bus error
pdm (break 1) 3 ->
```

Fixing compiler and make errors

To fix compile-time errors:

1 To open the Error Browser, click on the Error Browser button in the Main Window.



2 To display the error, select the folder for the file.

If the debugger can associate an error with a specific file, then the messages appear under the folder for that file. Messages that are not associated with a particular file are listed under a folder labeled **MAKE***n*, where *n* is a unique number.

3 To display the code in the Source area, select the message. The line causing the error is highlighted in the Source area.

4 To edit the code, select the Edit symbol to the left of the message.

5 Fix the problem in your text editor and save the file.

6 Rebuild your executable with the **make** command.

7 Click on the **Build** button to load the new version of the executable.

Fixing errors that occur during execution

To fix run-time errors:

1 Examine your program at the break level. Refer to "Using Workspace break levels" on page 160.

2 Edit the code. Refer to "Editing source code" on page 145.

3 Fix the problem in your text editor and save the file.

4 Rebuild your executable with the **make** command:

```
pdm -> make target
```

5 Click on the **Build** button to load the new version of the executable.

Removing errors from the Error Browser

To remove messages from the Error Browser:

1 Open the Error Browser.

2 Select the folder for the file.

3 Select the message.

4 Select the **Remove Selected** button in the Button panel.

Example

“Correcting compiler and make errors” on page 107 provides an example of using the Error Browser.

See also

"Editing source code" on page 145

"Using Workspace break levels" on page 160

"Moving in the execution stack" on page 166

Task: Setting breakpoints and watchpoints

You can set **breakpoints**, which always stop execution, **conditional breakpoints**, which stop execution only if a condition is true, and **watchpoints**, which stop execution when the value of an expression changes.

When you set a breakpoint:

- The debugger creates a debugging item. Debugging items are numbered from 1 to *n* and include breakpoints, action points, and data structures that are displayed in the Data Browser. You can display the current list of debugging items and delete either specific debugging items or all debugging items. For more information, refer to "Examining and deleting debugging items" on page 158.

- The Source area displays a Breakpoint symbol to the left of the line number:

Breakpoint symbol

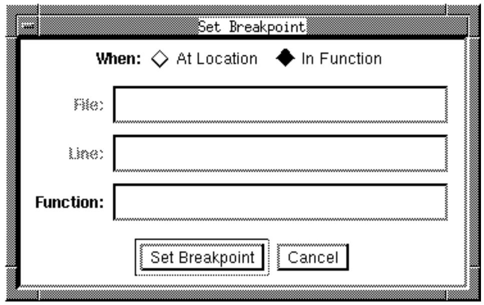
When execution reaches a breakpoint, the debugger stops execution, creates a break level in the Workspace, and lists the current line of code in the Source area.

Setting breakpoints at a line number or in a function

You can set breakpoints at a line number or in a function. From the Main Window, you can use the following methods to set breakpoints:

- Select the line number.

- Display the **Debug** menu and select **Set Breakpoint**. If you want to set a breakpoint at a location, specify the filename and line number in the dialog box. If you want to set a breakpoint at a function, specify the function name in the dialog box. The following illustration shows the Set Breakpoint dialog.



- Display the **File Options** pop-up menu and select **Set Breakpoint**. If you want to set a breakpoint at a location, specify the filename and line number in the dialog box. If you want to set a breakpoint at a function, specify the function name in the dialog box.

- Display the **Line Number** pop-up menu and select **Set Breakpoint Here**.

- Use the **stop** command:

```
pdm -> stop at line
pdm -> stop in function
```

To set a breakpoint on the line where execution is currently stopped, use the stop command with no arguments:

```
pdm -> stop
```

Setting conditional breakpoints

You can use the **stop** command to set conditional breakpoints by specifying a boolean expression as the condition.

- To set breakpoints on the line where execution is currently stopped.

```
pdm -> stop if cond
```

- To set breakpoints on a specific source line:

```
pdm -> stop at line if cond
```

- To set breakpoints in a specific function:

```
pdm -> stop in func if cond
```

Setting breakpoints in shared libraries

To set a breakpoint in a C library function, such as **printf()**:

1 Set a breakpoint in **main()**:

```
pdm -> stop in main
stop (1) set at "main.C":n, main().
```

2 Run the executable:

```
pdm -> run
Executing: program
```

3 Set the breakpoint in the C library function:

```
pdm (break 1) -> stop in function
stop (2) set at 0xf76c9514, function().
```

4 Continue execution to reach the breakpoint:

```
pdm (break 1) -> cont
Stopped in function: 'function'. No source file info.
```

You can then use machine debugging commands to examine the C library function. Refer to "Debugging machine instructions" on page 186 for more information.

Once you set a breakpoint in a C library function this way, the debugger retains the breakpoint in subsequent runs of the program.

Setting breakpoints in machine code

If you are debugging code that has not been compiled with the **-g** switch, you can set breakpoints in machine code. With the **stopi** command, you set breakpoints on the current line where execution is stopped or at a specific address:

```
pdm -> stopi
pdm -> stopi at address
```

For more information, refer to "Debugging machine instructions" on page 186.

Setting breakpoints in inline functions

To set a breakpoint in an inline function, you must compile the executable with the **+d** switch, which does not expand inline functions as inline but as static functions.

Setting watchpoints

You can use the **watch** *expr* command in **gdb** to set watchpoints. For more information about using **gdb** in the CenterLine-C++ debugger, refer to **gdb** on page 222 and **gdb_mode** on page 223.

Example

In this example, you set a conditional breakpoint in the Bounce program.

NOTE If you have not yet set up the CenterLine-C++ examples directory, refer to “Setting up the examples directory” on page 9 for instructions.

---

1 Build and load the Bounce program:

```
pdm -> cd ~/c++examples_dir
pdm -> make bounce
pdm -> debug bounce
```

2 List **shapes.C**:

```
pdm -> list shapes.C
```

3 In the Source area, scroll to the **DrawableShape::drawMove** function on line 102.

4 In the Workspace, set a breakpoint in this function that occurs only if the value of count is 250:

```
pdm -> stop in DrawableShape::drawMove if count == 250
stop(1) set at “shapes.C”:104, DrawableShape::drawMove(int).
```

5 Run the Bounce program by selecting the **Run** button in the Button panel. The Bounce window appears, and the Rectangle bounces. When the breakpoint is reached, the rectangle stops bouncing and a break level is generated in the Workspace.

6 Check the value of the **count** variable:

```
pdm -> print count
(int) 250
```

7 Continue execution of the Bounce program by selecting the **Continue** button in the Button panel.

See also

stop on page 252

stopi on page 254

"Setting actions" on page 155

"Examining and deleting debugging items" on page 158

"Using Workspace break levels" on page 160

Task: Setting actions

You can set an action on a line or in a function with the **when** command. An action performs a set of Workspace commands when execution reaches the line or function. To set an action:

1. Enter the **when** command with the desired arguments in the Workspace:

```
pdm -> when                 ...sets action on current line
pdm -> when at line         ...sets action on line
pdm -> when in func         ...sets action in function
```

2. Optionally, add a boolean condition to the arguments. This boolean condition must be met for the action to be carried out:

```
pdm -> when if cond
pdm -> when at line if cond
pdm -> when in func if cond
```

3. Enter one Workspace command per line at the **when** prompt:

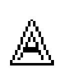
```
pdm -> when [arguments]
when -> command
when -> command
when -> ...
```

4. Since all actions include an implicit breakpoint (**stop** command), add the **cont** command as the last command if you want your program to continue execution after carrying out the action:

```
pdm -> when [arguments]
when -> command
when -> ...
when -> command
when -> cont
```

5. End the action by entering a period (**.**) or **end** as the last command or by pressing Control-d.

When you set an action, the debugger creates a debugging item and displays the action symbol to the left of the line number in the source area:



Action symbol

When an action is triggered during execution of your program, the debugger displays the Execution symbol on the line in the Source area, stops execution of the program, creates a break level, and evaluates the commands in the action.

Example

In this example, you set two different kinds of actions in the Bounce program.

NOTE If you have not yet set up the CenterLine-C++ examples directory, refer to “Setting up the examples directory” on page 9 for instructions.

---

1 Build and load the Bounce program:

```
pdm -> cd ~/c++examples_dir
pdm -> make bounce
pdm -> debug bounce
```

2 List mainfixed.C:

```
pdm -> list mainfixed.C
```

3 Set an action on line 15 that prints the value of ***r**, which is the rectangle being bounced.

```
pdm -> when at 15
when (1) set at “mainfixed.C”:15, main().
Type commands to be executed (one per line). Finish
by typing a single “.” or “end”
when -> print r
when -> cont
when -> .
pdm ->
```

4 Set another action in the **DrawableShape::drawMove** function that increments the count variable when it reaches 300:

```
pdm -> when in DrawableShape::drawMove if count ==
300
when (2) set at “mainfixed.C”:15, main().
Type commands to be executed (one per line). Finish
by typing a single “.” or “end”
when -> dump
when -> cont
when -> .
pdm ->
```

5 Select the **Run** button in the Button panel.

As the Bounce program executes, it carries out the action.

See also

"Examining and deleting debugging items" on page 158

"Using Workspace break levels" on page 160

when on page 260

Task: Examining and deleting debugging items

Each time you set a breakpoint, set an action point, or display a data structure, the debugger sets a debugging item.

You can examine debugging items to see everything that affects the execution of your program. Although you might not think that data items affect execution of your program, they actually do. At each break in execution, the debugger updates all data items. For more information, refer to “Updating data items” on page 172.

Examining debugging items

You can use two different methods to examine debugging items:

- Display the **Examine** menu and select **Status**.
- Enter the **status** command in the Workspace.

With either method, the Workspace lists all the debugging items:

```
pdm (break 1) 6 -> status
(5) stop at “main.C”:17 /* main() */
(6) display r::r
(7) display *r::r
```

Deleting debugging items

You can delete debugging items in a variety of different ways:

Main Window	From the **Debug** menu, display the **Delete** submenu, which displays all debugging items currently defined. Select the item to be deleted.
Source area	Select the Breakpoint symbol itself. *or* Display the **File Options** pop-up menu and then the **Delete** submenu. Select the item to be deleted.
Workspace	Display the **Workspace Options** pop-up menu and then the **Delete** submenu. Select the item you want to delete. *or* Use the **delete** command.

Data Browser — Select the data item and click on the **Remove Selected** button in the button panel.

or

Select the data item. Display the **Graph** menu and select **Remove Selected**.

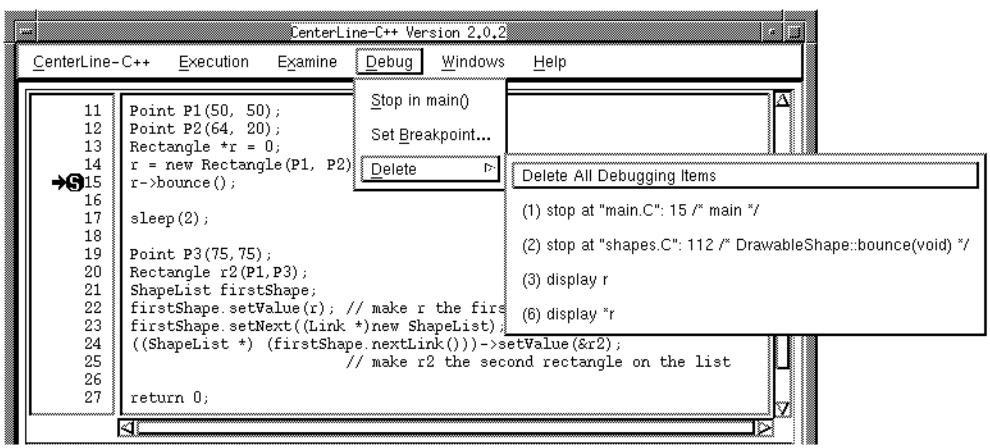
or

Select the **Clear** button to remove all data items.

Example

If you have some debugging items set, you can delete all debugging items as follows:

1. Display the **Debug** menu and then the **Delete** submenu, which lists all current debugging items.
2. Select **Delete All Debugging Items**.



See also

delete on page 214

status on page 248

Task: Using Workspace break levels

Generating a break level

When your program begins executing in the debugger, it is at the top level of execution in the Workspace. To create a different level of execution, called a **break level**, you can interrupt your program's execution in one of the following ways:

- Press Control-c while your program is executing.
- Set a breakpoint before running your program. When execution reaches a breakpoint in the program, a break level is created.
- Cause a signal to occur while your program is executing. If the signal is not handled by your program, a break level is created.

The current break level is indicated by the Workspace prompt:

```
pdm (break 1) 8 ->
```

NOTE In the debugger, there is only one break level.

Examining your program at a break level

Working at a break level allows you to preserve the flow of execution at one point, work with a different flow of execution, and then return to continue the previous flow of execution.

You can use the following Workspace commands to explore your program at a break level. Many of these commands are available from the **Examine** menu, **Execution** menu, and Button panel in the Main Window.

cont	Continues execution from a break location.
delete	Deletes an existing debugging item on the current line.
down	Moves the current scope location down the execution stack.
dump	Displays all local variables.
edit	Invokes your editor, positioned at the current line.
file	Displays and sets the current list location.

next	Executes the next line; does not enter functions.
print	Prints the value of variables or expressions.
step	Steps execution by statement, entering functions.
stepout	Continues execution until the current function returns.
stop	Sets a breakpoint.
up	Moves the current scope location up the execution stack.
whatis	Displays all uses of a name for a function, data variable, tag name, enumerator, type definition, or macro definition.
when	Specifies statements to execute when execution triggers the action.
where	Displays the execution stack.
whereami	Displays the current break and scope locations.
whereis	Lists the defining instance of a symbol. If the symbol is an initialized global variable, **whereis** also indicates the location at which it is initialized.

For detailed information on how to use each of these commands, see the Man Browser or Chapter 7, “Command Reference,” on page 201.

While at a break level, you can examine the state of data structures that are currently in scope by using the Data Browser. For more information, refer to "Examining data structures" on page 169.

Changing locations in break levels

While at a break level, the debugger maintains several distinct locations. A **location** is a specific line number of a source file.

Break location — The location at which execution was stopped and is resumed when you continue. The values of program variables are determined by their scope at the break location. The break location is indicated in the Source area by the Execution symbol:

Execution symbol

Scope location — The current location in the call stack. When a break level is created, the scope location is set to the break location. The scope location, however, can change in response to your actions at a break level. For example, you can use the **up** and **down** commands to move between stack frames in the execution stack. The scope location is indicated in the Source area by the Scope symbol:

Scope symbol

Source location — The default location used by commands that handle source code files, such as **list** and **edit**. When a break level is created, the source location is set to the break location. When the scope location changes, the source location is set to the new scope location. The **file** and **list** commands also change the source location.

Examples

See "Moving in the execution stack" on page 166 and "Examining data structures" on page 169 for examples of how you can use Workspace break levels.

See also

"Running, continuing, and stepping" on page 163

"Moving in the execution stack" on page 166

"Examining data structures" on page 169

Task: Running, continuing, and stepping

Running your program

You can run your program using any of these methods:

- Display the **Execution** menu and select **Run**.
- Select the **Run** button in the Button panel.
- Issue the **run** command in the Workspace:

```
pdm -> run
pdm -> run arguments
```

With any of the methods, the debugger executes **main()** after initializing all variables and processing any command-line arguments. If you specify arguments with the **run** command, the debugger uses these arguments each subsequent time you use the **Run** menu command, **Run** button, or the **run** command without arguments.

You can use the **rerun** command to run your program with different arguments instead of passing different arguments to **run**. For example, if you were testing your program with two different files as input arguments, you could issue **run** *file1* and **rerun** *file2* and then alternate between **run** and **rerun** as needed.

Continuing from the break location

If your program is stopped, you can continue execution using one of the following methods:

- Display the **Execution** menu and select **Continue**.
- Select the **Continue** button in the Button panel.
- Issue the **cont** command in the Workspace.

The **cont** command has additional arguments for continuing from a specific line number or with a specific signal. For more information, refer to **cont** on page 210.

Resetting to the top level

To transfer control from the break level to the top level, issue the **reset** command from the Workspace:

```
pdm -> reset
Resetting to top level.
```

There are two other situations where control is transferred from the break level to the top level:

- If you rebuild your executable with the **make** command, and then reload the new executable with the **build** command, the debugger resets to the top level.
- If you delete all breakpoints and then continue execution with the **cont** or **run** commands, the program runs to completion, and control returns to the top level.

Stepping through your program

You can step through your program from a break level by using the **step**, **next**, and **stepout** commands in the Workspace. The **step** and **next** commands are also available on the **Execution** menu and in the Button panel.

- The **step** command continues execution until the next statement is reached. If execution is stopped on a line containing multiple statements, **step** executes the next statement only. Also, **step** enters functions. To step through more than one statement, you can pass a numeric argument to the **step** command:

  ```
  pdm -> step n
  ```

- The **next** command executes all statements on the current source code line. It does not enter functions with one exception—if the function contains a breakpoint that is triggered, **next** enters the function and stops at the breakpoint. To step through more than one line, you can pass a numeric argument to the **next** command.

  ```
  pdm -> next n
  ```

- The **stepout** command continues execution until the current function returns. This command can be useful if you accidentally entered a function by using **step** instead of **next**.

  ```
  pdm -> stepout
  Run till exit from #n function at filename:line
  ```

When you step through your program, the Execution symbol in the Source area shows the current line. In addition, data items in the Data Browser are automatically updated at every break in execution (see "Examining data structures" on page 169).

NOTE If you are stepping through your code and have many data items displayed in the Data Browser, the time spent in updating the data items at each break in execution can degrade performance. You can improve performance by minimizing the number of items you have displayed, or by dismissing the Data Browser.

See also

"Debugging machine instructions" on page 186

cont on page 210

next on page 233

rerun on page 238

reset on page 239

run on page 240

step on page 249

stepout on page 251

Task: Moving in the execution stack

The **execution stack** consists of all the functions that are in the process of execution. Each function has a stack frame. Stack frames are numbered, beginning at 0, as they are placed on the execution stack.

Displaying the execution stack

To display the stack, use the **where** command:

```
pdm (break 1) -> where
```

The execution stack is displayed starting from the location where execution has stopped in the current break level. For example, in the following stack, execution is stopped in **makeTable** (#0).

```
#0 makeTable (rows=0x8ae0, cols=0x8ed0, length=500, width=600, height=600,
 iwidth=64, iheight=20, margin=0) at table.c:45
#1 0x3258 in DrawableShape::createTable (this=0x8ab0) at shapes.C:132
#2 0x30bc in DrawableShape::doDraw (this=0x8ab0) at shapes.C:89
#3 0x31d0 in DrawableShape::bounce (this=0x8ab0) at shapes.C:112
#4 0x2b90 in main () at mainfixed.C:15
```

Moving in the execution stack

When a break level is generated, the break location and the scope location are identical—all variables, types, and macros are scoped to the point at which execution was interrupted. You can change the scope location to another function on the execution stack with the **up** and **down** commands.

When you issue **up** or **down** to move in the execution stack, the debugger displays the scope location (now different from the break location) in the Source area using the Scope symbol.

Scope symbol

With the **up** command, you move to the stack frame with the next higher number. If the scope location is currently at #1, issuing **up** moves to #2. The converse is true with the **down** command; it moves the scope location to the next lower numbered stack frame.

Displaying the current stack location

The **whereami** command shows where you currently are in the execution stack. The **whereami** command displays the scope location in the Source area (scrolling the display if necessary) and, if the scope location is different from the break location, displays the break location in the Workspace.

Example

In this example, you examine the execution stack of a simple program that contains a static constructor, the **static.C** program, which is provided in the **c++examples_dir** directory.

NOTE If you have not yet set up the CenterLine-C++ examples directory, refer to “Setting up the examples directory” on page 9 for instructions.

---

1 Change to the CenterLine-C++ examples directory:

```
pdm -> cd ~/c++examples_dir
```

2 Build, load, and list the **static.C** program by issuing the following commands in the Workspace:

```
pdm -> make static
...
pdm -> debug static
Debugging program ‘static’
pdm -> list static.C
Listing file c++_examples/static.C, line 1
```

3 Select the line number on lines 7 and 8 to set breakpoints on those lines.

4 Select the **Run** button in the Button panel.

The **static** program executes and stops in **main** on line 8.

5 To examine the stack, use the **where** command:

```
pdm (break 1) 20 -> where
#0 main () at static.C:8
```

6 Select the **Continue** button in the Button panel.

The program stops on line 7 at the static constructor **foo**.

7 Examine the stack again.

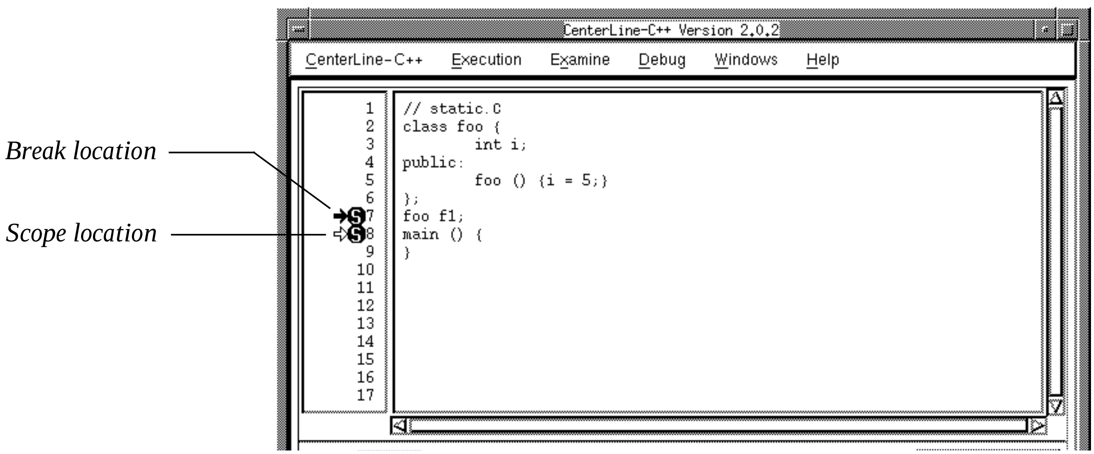
```
pdm (break 1) 8 -> where
#0 __sti__static_C_main_ () at static.C:7
#1 0x2354 in _main ()
#2 0x22a0 in main () at static.C:8
```

As you can see, the top-most function (#0) in the execution stack is the static constructor.

8 Move up the stack to the static constructor:

```
pdm (break 1) 22 -> up
Scoping to _main() at “??”, pc = 2354
pdm (break 1) 23 -> up
Scoping to main() at “static.C”:8
pdm (break 1) 24 ->
```

The Main Window shows that the scope location is now different from the break location:



9 To display this same information in the Workspace, use the **whereami** command:

```
pdm (break 1) 10 -> whereami
Break Location: __sti__static_C_main_() at
“static.C”:7
Scope Location: main() at “static.C”:8
```

See also

down on page 217

up on page 257

where on page 262

whereami on page 263

Task: Examining data structures

Once you have loaded your executable into the debugger or are executing it in the debugger, you can examine data structures. You can only examine data structures that are currently in scope:

- When your program is at the top level, you can examine only global data.
- When your program is at a break level, you can examine global and local data. For example, you can use the **whatis** command, the **print** command, the **dump** command, and the Data Browser.

Each time you display a data structure in the Data Browser, the debugger sets a debugging item. For more information about debugging items, refer to "Examining and deleting debugging items" on page 158.

Displaying data structures in the Workspace

To display all uses of a name for a function, data variable, tag name, enumerator, type definition or macro definition, use the **whatis** command:

```
pdm -> whatis name
```

To display the defining instance of a symbol, use the **whereis** command:

```
pdm -> whereis name
```

If the symbol is an initialized global variable, **whereis** also displays the location at which the variable is initialized.

To display the name and value of each variable that is local to the current scope location, use the **dump** command:

```
pdm -> dump
```

To print the value of a specific variable, use the **print** command:

```
pdm -> print variable
```

You can improve performance when displaying data structures by setting the **class_as_struct** option as described on page 192.

Assigning values to variables

You can use the **assign** and **set** commands, which are functionally equivalent, to evaluate an expression and assign it to a variable. By assigning a value to a variable in the Workspace, you can directly manipulate values in code that you are debugging. The assign and set commands take the following form:

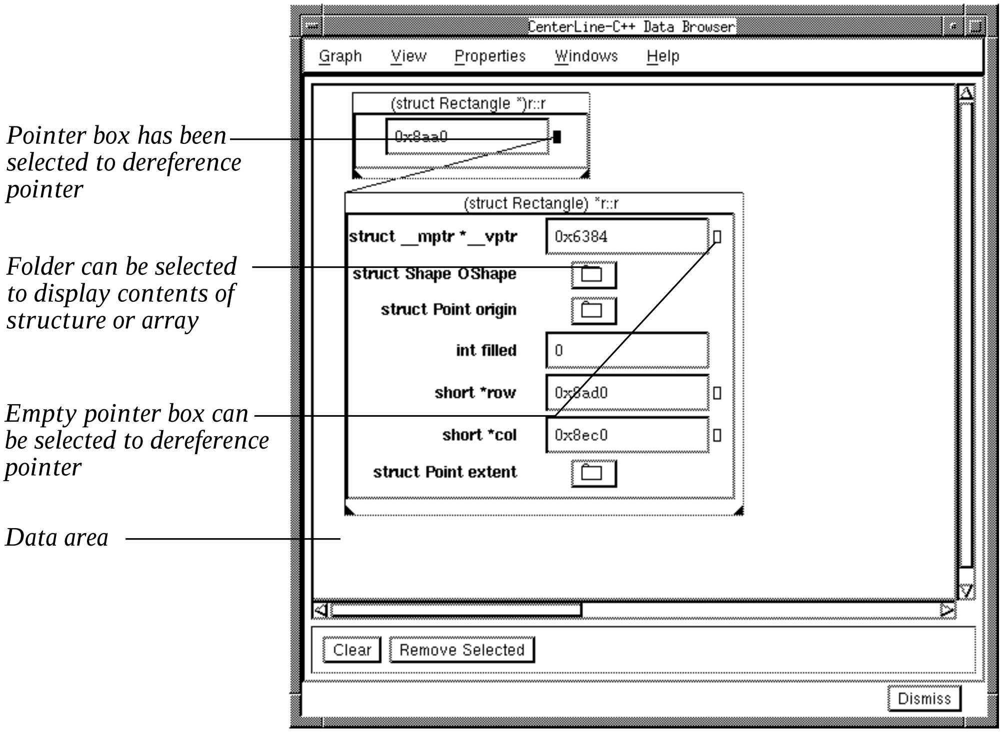
```
pdm -> assign variable = expression
pdm -> set variable = expression
```

Displaying data structures in the Data Browser

You use the Data Browser to display a graphical representation of any data structure. This graphical representation is called a **data item**. To display the data item for a data structure, you can:

Main Window	From any debugger window, select a variable that is currently in scope. Display the **Examine** menu and select **Display<Selection>**.
Data Browser	Display the **Graph** menu and select **New Expression**. In the dialog box, enter the variable or expression and select the **Data Browse** button.
Workspace	Use the **display** command.

The following illustration shows the Data Browser with two data items displayed in the Data area: a pointer and the structure it references.



As you can see from the figure, a pointer is indicated by a pointer box, which has different fills to indicate whether you can dereference it:

- If the box is empty, you can dereference the pointer.
- If the box is filled, the pointer has been dereferenced and a line connects to the referenced data item. A dotted line (not shown in the figure) connecting two data items indicates a pointer that points to data *inside* a structure rather than to the top of the structure.
- If the box contains an X, the pointer is invalid (null). This type of box is not shown in the figure.

Structures and arrays are indicated by folders, which you can select to display their contents.

Updating data items

When data has changed, data items are automatically updated at every break in execution and each time you use the **assign** and **set** commands in the Workspace.

NOTE If you are stepping through your code and have many data items displayed in the Data Browser, the time spent in updating the data items at each break in execution can degrade performance. You can improve performance by minimizing the number of items you have displayed, or by dismissing the Data Browser.

Working with data items

You can display as many data structures as you want in the Data Browser. If you display a number of data items, you can use the scrollbars to view them.

You can select data items and manipulate them with menus. To select a data item, click the Left mouse button on it. The data item should have a bolder outline when it is selected. In addition to selecting items individually, you can select groups of them by dragging the mouse pointer to enclose the desired items with a bounding box. You can also move selected items by dragging them where you want them.

The Data Browser provides three menus for manipulating *selected* data items: the **Graph** menu, the **View** menu, and the data item pop-up menu. To display the data item pop-up menu:

1 Move the mouse pointer over the title bar of a data item.

2 Press and hold the Right mouse button.

You can manipulate data items in a number of different ways:

Select	Selects one or more data items, depending on the scope specified (**Ancestors**, **Parents**, **Children**, **Descendants**).
Unselect	Unselects one or more data items, depending on the scope specified (**Ancestors**, **Parents**, **Children**, **Descendants**).
Iconify	Displays only the name and not the content of the data structure.
Deiconify	Reverts to displaying both the name and content of the data item (its original size before you iconified it).

Raise	Moves the data item in front of all other data items.
Lower	Moves the data item behind all other data items.
Zoom	Opens all folders in the data item and resizes it.
Shrink	Closes all folders in the data item.
Remove	Removes the data item.
Properties	Changes the way the data item is displayed.

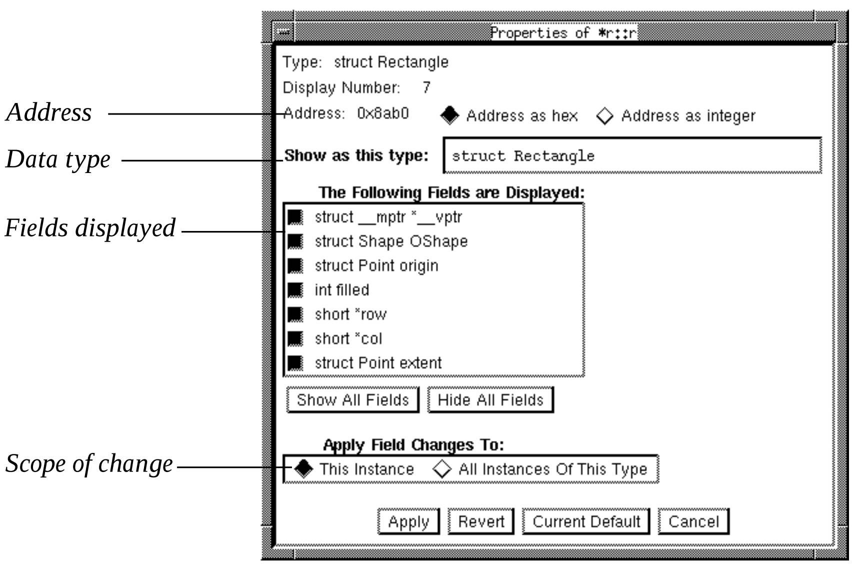
Changing the properties of a data item

You can change the properties of a data item to display the data structure differently. To do so:

1 Select the data item.

2 Display the **Properties** menu and select **Item Properties**.

The Properties dialog box appears:



3 If you prefer to display the address as an integer instead of in hexadecimal (default), select the **Address as Integer** radio button.

4 If you want to display this data structure using a different data type, enter it in the **Show as this type** field.

5 If you want to hide any of the fields in the data structure, deselect the button beside each field in the list of displayed fields.

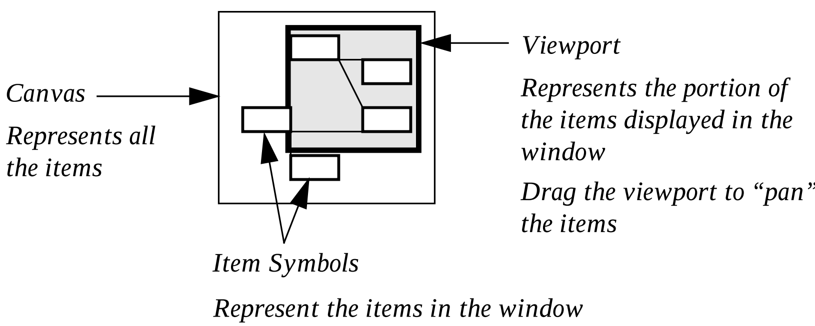
6 If you want to apply your changes to all data items with this same data type, select the **All Instances of This Type** button.

7 Select the **Apply** button.

If you make a mistake, you can select **Revert** to revert to the last settings that were applied or **Current Default** to revert to the default settings for this kind of data type.

Customizing the Data Browser

As an alternative to using scrollbars, you can customize the Data Browser to use a panner. A **panner** contains a canvas representing all the data items you have displayed and a viewport that represents what is currently shown in the Data area:



To use a panner, put the following resource in your **.Xdefaults** file:

```
CenterLine-C++*usePanner: True
```

Examples

In this example, you learn how to display an array of characters in the Data Browser.

NOTE If you have not yet set up the CenterLine-C++ examples directory, refer to “Setting up the examples directory” on page 9 for instructions.

---

1 Go to the CenterLine-C++ examples directory and make the **chararray** target:

```
pdm -> cd ~/c++examples_dir
pdm -> make chararray
```

2 Load **chararray**.

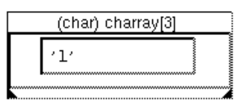
```
pdm -> debug chararray
```

3 Set a breakpoint on line 6 (**return 0;**).

4 Select the **Run** button.

5 Display the fourth element (3) in the array with the **display** command:

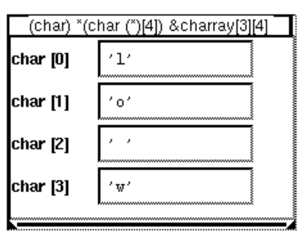
```
pdm -> display charray[3]
display(1) set on expression ‘charray[3]’
```



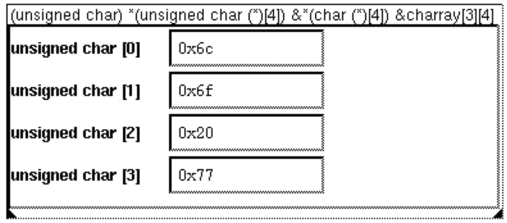
6 In the Data Browser, select the data item, display the **Properties** menu, and select **Item Properties**.

7 Change the **Show as this type** field from **char** to **char[4]**.

8 Press Return or select the **Apply** button.



9 Repeat Steps 6, 7, and 8, but change the type to **unsigned char[4]**.



10 In the Main Window, select the **Continue** button to complete execution of the program.

See also

"Examining and deleting debugging items" on page 158

assign on page 204

display on page 216

dump on page 218

print on page 235

set on page 242

whatis on page 259

whereis on page 265

Task: Handling signals

By default, the debugger passes the following signals to your program and catches all other signals.

SIGALRM
SIGURG
SIGCONT
SIGCHLD
SIGPOLL
SIGVTALRM
SIGPROF
SIGWINCH

Passing signals to your program

To ignore a signal and pass it to your program:

1 Make sure your program defines a handler for the signal.

2 Issue the **ignore** command and specify the signal name or number.

```
pdm -> ignore {signal_number | signal_name}
```

To display the signals currently being ignored, issue **ignore** without arguments.

Catching signals

To catch a signal, issue the catch command and specify the signal name or number.

```
pdm -> catch {signal_number | signal_name}
```

When the debugger encounters a signal that is currently being caught, it stops execution of your program and generates a break level.

To display the signals currently being caught, issue **catch** without arguments.

See also

"Debugging an executable with a corefile" on page 178

catch on page 207

ignore on page 226

Task: Debugging an executable with a corefile

Generating corefiles

When you run your program with the debugger and a signal occurs, a corefile might or might not be generated, depending on how the debugger handles the signal. If you run your program from the debugger and it catches the signal, a corefile is not generated. If the debugger ignores the signal and passes it to your program, a corefile is generated. For more information on signals, refer to "Handling signals" on page 177.

You can also generate a corefile by running your program from the shell.

Loading corefiles

You can load a corefile using the **debug** command or the **centerline-c++** shell command (when you start up the debugger). Refer to "Starting up the debugger" on page 120 and "Loading files for debugging" on page 142 for information on loading corefiles.

When you load an executable along with a corefile into the debugger, the Execution symbol in the Source area indicates the source line at which execution stopped, and the Workspace indicates the signal that caused the program to terminate.

Debugging corefiles

When you have loaded a corefile, you can examine all portions of the program up to the source line marked by the Execution symbol. You can examine data as well as the execution stack.

You cannot, however, continue execution of the program or single step. If you run the program while a corefile is loaded, the debugger discards the corefile and runs the executable instead. To examine the corefile again, you must reload it along with the executable.

Example

In this example, you generate a corefile and examine it with the debugger.

NOTE If you have not yet set up the CenterLine-C++ examples directory, refer to “Setting up the examples directory” on page 9 for instructions.

---

1 Go to the CenterLine-C++ examples directory and make the **tutorial_core** target:

```
pdm -> cd ~/c++examples_dir
pdm -> make tutorial_core
```

2 Load **tutorial_core** and run it.

```
pdm -> debug tutorial_core
pdm -> run
Executing c++examples_dir/tutorial_core

Program received signal 11, Segmentation fault
pdm (break 1) ->
```

Since the debugger generated a break level, it has caught the SIGSEGV signal, and a corefile was not generated.

3 Ignore the SIGSEGV signal and rerun the program to generate the corefile:

```
pdm (break 1) -> ignore 11
pdm (break 1) -> run
Restting to top level.
Executing c++examples_dir/tutorial_core
Program terminated with signal 11, Segmentation fault
The inferior process no longer exists.
Resetting to top level.
pdm ->
```

4 Load **tutorial_core** and the corefile:

```
pdm -> debug tutorial_core core
debug: Deleting all debugging items.
Debugging program ‘tutorial_core’ (previous program ‘tutorial_core’)
Core was generated by ‘tutorial_core’.
Program terminated with signal 11, Segmentation fault.
#0  DrawableShape::bounce (this=0x0) at shapes.C:112
    112   doDraw();
```

The debugger checks the executable against the corefile to make sure that they match, displays the Execution symbol on the source line that generated the signal, and indicates the signal that generated the corefile.

5 To display all local variables in **DrawableShape::bounce()**, display the **Examine** menu and select **Dump**. The Workspace indicates there are no local automatic variables.

6 To move up in the execution stack and examine **main()**, select the **Up** button in the Button panel. The Source area displays the Scoping symbol on line 15.

7 To display the execution stack:

```
pdm -> where
#0  DrawableShape::bounce (this=0x0) at shapes.C:112
#1  0x2b74 in main () at main.C:15
pdm ->
```

8 To display all local variables in **main()**, display the **Examine** menu and select **Dump**. The Workspace displays **P1**, **P2**, and **r**:

```
pdm -> dump
Formals of ‘main’:
No Formals
Automatics of ‘main’:
P1 = (struct Point) = {
 int x = 50;
 int y = 50;
 }
P2 = (struct Point) = {
 int x = 64;
 int y = 20;
 }
r = (struct Rectangle *) 0x0
_result = (int) 0
```

See also

"Starting up the debugger" on page 120

"Loading files for debugging" on page 142

"Using Workspace break levels" on page 160

"Examining data structures" on page 169

"Handling signals" on page 177

Task: Debugging a running process

The debugger can run one process under its control. You can run an executable that you have loaded with the **debug** command, or you can attach to a running process. You cannot do both simultaneously.

Attaching to processes

You can attach to a running process in several different ways:

- Start the debugger with the **centerline-c++** command and specify the process ID of a running process along with its corresponding executable. Refer to "Starting up the debugger" on page 120 for information.

- Load an executable for debugging with the **debug** command and specify the process ID of a running process along with its corresponding executable. Refer to "Loading files for debugging" on page 142 for information.

- Use the **attach** command and specify the process ID of a running process. The running process can match the executable that is loaded or be an unrelated process.

  ```
  pdm -> attach process_id
  ```

When you attach to a process, the debugger suspends its execution, as if the process were at a breakpoint. The kind of debugging you can perform depends on whether the process was compiled with **-g** and the point during execution at which you attach to the process.

- If the process was not compiled with **-g**, you can only use machine debugging to debug the process. For more information, refer to "Debugging machine instructions" on page 186.

- If you attach to the process at a point where it is executing in code that was not compiled with **-g** (such as system libraries), you can only use machine debugging techniques until you enter a routine that was compiled with **-g**. Refer to "Moving in the execution stack" on page 166 and "Debugging machine instructions" on page 186 for more information.

- If you attach to a process that was compiled with **-g**, you can use all the features of the debugger to debug it.

If you use the **step** command after attaching to a program that was not compiled with **-g** (or a program that is currently executing in a routine that was not compiled with **-g**), execution continues until the current function completes because there is no source line information. To single step an instruction, you must use the **stepi** command.

If you issue the **run** command while the debugger is attached to a process, the debugger terminates the process and runs a new process with the executable that was previously loaded with the **debug** command.

Detaching from processes

To detach a process from the debugger's control, use the **detach** command:

```
pdm -> detach
```

The process continues executing until completion. You can now attach to or run a different process.

Debugging multiple processes

The CenterLine-C++ debugger does not have any special support for debugging multiple processes. We recommend that you use ObjectCenter to debug multiprocess programs. It provides a built-in macro for invoking an additional ObjectCenter process when forking a child process.

Example

In this example, you run a process from a shell and attach to it with the debugger.

NOTE If you have not yet set up the CenterLine-C++ examples directory, refer to "Setting up the examples directory" on page 9 for instructions.

---

1 Go to the CenterLine-C++ examples directory and make the **tutorial_attach** target:

```
pdm -> cd ~/c++examples_dir
pdm -> make tutorial_attach
```

2 Load **tutorial_attach**.

```
pdm -> debug tutorial_attach
```

3 From a shell, run the **tutorial_attach** program in the background, and notice the process ID that is assigned to it.

```
% tutorial_attach &
[n] process_id
```

4 From the debugger, attach to the process ID that was assigned to **tutorial_attach**.

```
pdm -> attach process_id
Attaching program ‘c++examples_dir/tutorial_attach’, pid process_id
Reading symbols from libX11.so.4.3...done.
Reading symbols from libC.so.2.0...done.
Reading symbols from /libc.so.1.6...done.
Stopped in function: ‘sigpause’. No source file info.
pdm (break 1) ->
```

The debugger stops the process and creates a break level. In this case, **tutorial_attach** was executing in **sigpause**, which is a library routine that was not compiled with **-g**.

5 To display the current execution stack:

```
pdm (break 1) -> where
#0 0xf76b2898 in sigpause ()
#1 0xf76c965c in sleep ()
#2 0x2b90 in main () at mainwait.C:16
```

The stack shows that execution was stopped in the **sigpause** routine, which was called from the **sleep** routine. Line 16 of **mainwait.C** called the **sleep** routine.

6 To move the scope location to **mainwait.C**, use the **up** command:

```
pdm (break 1) -> up
Scoping to sleep() at “??”, pc = f76c965c
pdm (break 1) -> up
Scoping to main() at “mainwait.C”:16
```

The Source area now displays the source code for **mainwait.C**, and the Scope symbol points to line 16.

7 To continue execution until **sleep** returns to **main**, you can use the **step** command. When you use the **step** command in routines that were not compiled with **-g** (and do not have

debugging symbols), the debugger continues execution until the function returns.

```
pdm (break 1) -> step
Current function has no line number information.
Single stepping until function exit.
Stopped in function: ‘sleep’. No source file info.
pdm (break 1) -> step
Current function has no line number information.
Single stepping until function exit.
```

After a short pause, the Execution symbol appears at line 18 of **mainwait.C**.

8 To detach from the process:

```
pdm (break 1) -> detach
Detaching program: c++examples_dir/tutorial_attach pid process_id
pdm (break 1) ->
```

The process continues execution. The bounce window appears, and a rectangle bounces in it.

9 Reset the Workspace:

```
pdm (break 1) -> reset
Resetting to top level.
pdm ->
```

See also

attach on page 205

detach on page 215

Task: Debugging machine instructions

If a program or library was not compiled with **-g**, you can still debug its machine instructions using the **listi**, **stopi**, **stepi**, and **nexti** commands.

NOTE To debug machine instructions, you need to be familiar with the Assembler instructions for your platform's CPU as well as the registers used. Refer to the "Run-Time Organization" section of your CenterLine-C++ *Platform Guide* and the programming guide for your CPU.

Listing machine instructions

Use the **listi** Workspace command to list machine instructions in the Workspace. You can list machine instructions for:

- The current program counter (PC) address (if at a break level)
- A hexadecimal or octal address
- A range of hexadecimal or octal addresses
- A line number (if the executable was compiled with **-g**)
- A range of line numbers (if the executable was compiled with **-g**)
- A function
- An address expressed as a function name and offset

For example, if execution is stopped on the line that declares the **main** routine in the Bounce program, you can list the Assembler instructions for that source line in several different ways:

```
pdm -> listi                     (Current PC)
0x2b38       9 int main(void)
0x2b38 <main+8>: call 0x3430 <_main>
0x2b3c <main+12>: nop
pdm -> listi 0x2b38 0x2b3c       (Range of addresses)
0x2b38       9 int main(void)
0x2b38 <main+8>: call 0x3430 <_main>
0x2b3c <main+12>: nop
pdm -> listi 9                   (Source line number)
0x2b38       9 int main(void)
0x2b38 <main+8>: call 0x3430 <_main>
0x2b3c <main+12>: nop
```

Refer to **listi** on page 230 for complete syntax.

If the routine that you are listing has been compiled with **-g** and you list it with **listi**, the Workspace lists the source code interleaved with the Assembler code. For example:

```
0x2b38       9 int main(void)
0x2b38 <main+8>: call 0x3430 <_main>
0x2b3c <main+12>: nop

0x2b40      11 Point P1(50, 50);
0x2b40 <main+16>: add -16, %fp, %i4
0x2b44 <main+20>: mov %i4, %o0
0x2b48 <main+24>: mov 0x32, %o1
0x2b4c <main+28>: call 0x2c80 <__ct__5PointFiT1>
0x2b50 <main+32>: mov 0x32, %o2

0x2b54      12 Point P2(64, 20);
0x2b54 <main+36>: add -24, %fp, %i0
0x2b58 <main+40>: mov %i0, %o0
0x2b5c <main+44>: mov 0x40, %o1
0x2b60 <main+48>: call 0x2c80 <__ct__5PointFiT1>
0x2b64 <main+52>: mov 0x14, %o2
```

If the routine was compiled *without* **-g** and you list it with **listi**, the Workspace lists only the Assembler code. For example:

```
0x2b38 <main+8>: call 0x3430 <_main>
0x2b3c <main+12>: nop
0x2b40 <main+16>: add -16, %fp, %i4
0x2b44 <main+20>: mov %i4, %o0
0x2b48 <main+24>: mov 0x32, %o1
0x2b4c <main+28>: call 0x2c80 <__ct__5PointFiT1>
0x2b50 <main+32>: mov 0x32, %o2
0x2b54 <main+36>: add -24, %fp, %i0
0x2b58 <main+40>: mov %i0, %o0
0x2b5c <main+44>: mov 0x40, %o1
0x2b60 <main+48>: call 0x2c80 <__ct__5PointFiT1>
0x2b64 <main+52>: mov 0x14, %o2
```

Setting breakpoints in machine instructions

To set breakpoints on machine instructions, you can use the **stopi** command:

```
pdm -> stopi at address
```

When the byte at the specified address is modified, the debugger stops execution of the program. The *address* can be specified in hexadecimal, octal, or as a function plus offset.

Stepping through machine instructions

Use the **stepi** and **nexti** Workspace commands to step through machine instructions. **stepi** enters functions when they are called, but **nexti** does not enter functions.

If desired, you can step through more than one instruction at a time by specifying *number* as an argument to **stepi** or **nexti**:

```
pdm -> stepi number
pdm -> nexti number
```

Example

In the following example, the Bounce program has a bug in it, but it has not been compiled with **-g**, so you must debug it at the machine level. This example shows the machine instructions for the Sun SPARC platform; the machine instructions for other platforms differ.

NOTE If you have not yet set up the CenterLine-C++ examples directory, refer to “Setting up the examples directory” on page 9 for instructions.

---

1 Go to the CenterLine-C++ examples directory and make the **tutorial_corenog** target:

```
pdm -> cd ~/c++examples_dir
pdm -> make tutorial_corenog
```

2 Load **tutorial_corenog** and run it.

```
pdm -> debug tutorial_corenog
pdm -> run
Resetting to top level.
Executing: c++examples_dir/tutorial_corenog

Program received signal 11, Segmentation fault
pdm (break 1) ->
```

Since the debugger generated a break level, it has caught the SIGSEGV signal, and a corefile was not generated.

3 Ignore the SIGSEGV signal and rerun the program to generate the corefile:

```
pdm (break 1) -> ignore 11
pdm (break 1) -> run
Restting to top level.
Executing c++examples_dir/tutorial_corenog

Program terminated with signal 11, Segmentation fault
The inferior process no longer exists.
Resetting to top level.
pdm ->
```

4 Load **tutorial_core** and the corefile:

```
pdm -> debug tutorial_corenog core
debug: Deleting all debugging items.
Debugging program ‘tutorial_corenog’ (previous program ‘tutorial_corenog’)
Core was generated by ‘tutorial_corenog’.
Program terminated with signal 11, Segmentation fault.
#0 DrawableShape::bounce (this=0x0) at shapes.C:112
   112   doDraw();
pdm 7 ->
```

The debugger checks the executable against the corefile to make sure that they match, displays the Execution symbol on the source line that generated the signal, and indicates the signal that generated the corefile.

5 To show the instruction that generated the signal, use the **listi** command without any arguments, which displays the instruction at the current PC:

```
pdm -> listi
0x2da0 <bounce__13DrawableShapeFv+12>: ld [%i0], %g1
```

6 To see where the **DrawableShape::bounce** routine is called from **main**, move up the execution stack and display the current PC with the **listi** command:

```
pdm -> up
Scoping to main() at “mainorig.C”:15
pdm -> listi
0x29ac <main+60>: call 0x6184 <_GLOBAL_OFFSET_TABLE_+296>
```

See also

"Listing source code" on page 143

"Setting breakpoints and watchpoints" on page 150

"Running, continuing, and stepping" on page 163

listi on page 230

nexti on page 234

stepi on page 250

stopi on page 254

Task: Saving your debugging session

You can save a record of your debugging session in three different ways:

- Save your input history using the **history** command
- Save your complete debugging session, including input and output
- Save the input and output displayed in the Run Window

By default, the Workspace transcript is limited to 2000 lines, which limits the size of your input history or session transcript. To change the size of the Workspace transcript, use the **CenterLine-C++*workspaceTranscriptSize** resource. See the **X resources** entry in the Man Browser.

Displaying and saving your input history

To save your current input history in a file, redirect the output of **history** to a file:

```
pdm -> history #> filename
```

Saving your debugging session

To save your debugging session:

1. In the Workspace, display the **Workspace Options** pop-up menu and then the **Save Session to** submenu.
2. Select either **clc++.script** or **Other file**. If you select the latter, you specify the file in a file selection dialog box.

Saving the contents of the Run Window

To create a log in your current working directory of the contents of the Run Window during your debugging session, put the following resource in your X resources file. The filename of the log file is **XtermLog.***pid*, where *pid* is the process ID of the **clxterm** process (the process that controls the Run Window).

```
CenterLine-C++*RunWindow.logging: on
```

See also

history on page 225

centerline-c++ in the Man Browser

X resources in the Man Browser

Task: Customizing your startup file

The CenterLine-C++ debugger provides system-wide and local startup files that you can use for setting search paths and aliases at startup.

Setting up your .pdminit file

You can use the **.pdminit** startup file to set up aliases and search paths to be used across debugging sessions. When you start the debugger, it searches directories in the following order for a **.pdminit** file:

- Your current working directory. You can have different **.pdminit** files for use with different projects, providing the projects are stored in different directories.
- Your home directory.

To set up your **.pdminit** file:

1. Create a text file named **.pdminit** in your current directory or your home directory.

   If desired, you can give the file a different name from **.pdminit**. If you do, start the debugger with the **-s** *startup_file* switch and specify the pathname of the file. See “Selecting specific startup files from the command line” on page 192 for details.

2. Add **alias** commands for all the aliases you want to use across debugging sessions.

   ```
   alias name command
   ```

   For more information on aliases, refer to "Using aliases for Workspace commands" on page 136.

3. Add a **use** command for all the directories that the debugger should search through when loading (**debug** command), listing (**list** command), or editing (**edit** command) an executable.

   ```
   use directory...
   ```

4. Start **pdm**. It executes the commands in the **.pdminit** file.

Selecting specific startup files from the command line

When you start the debugger with the **centerline-c++** command, you can use the following switches to select or ignore specific startup files:

-s[=*filename*]	If *filename* is supplied, it is read instead of the local **.pdminit** file. If *filename* is not supplied, the debugger ignores the local **.pdminit** file.
-S[=*filename*]	If *filename* is supplied, it is read instead of the system-wide **.pdminit** file. If *filename* is not supplied, the debugger ignores the system-wide **.pdminit** file.

Setting the class_as_struct option

By default, **pdm** distinguishes between member functions and data members when it processes classes. You can improve performance by setting the **class_as_struct** option, which causes **pdm** to treat member functions and data members as data fields as in a C struct.

To make this the default behavior, add this line to your **.pdminit** file:

```
setopt class_as_struct
```

You can also use this command in the Workspace; if you do, it will take effect the next time you issue the **debug** command. To list current option settings, use the **printopt** command, and to set the **class_as_struct** option to false, enter this command:

```
unsetopt class_as_struct
```

Example

Here is a sample **.pdminit** file:

```
/* Define aliases for common commands. */

alias s         step
alias n         next

/* Specify search path. */

use . ../test ../src
```

See also

alias on page 203

use on page 258

centerline-c++ in the Man Browser

Task: Customizing buttons and commands

You can customize the menus in the Main Window in the following ways:

- Add, change, or delete buttons in the Button panel
- Create new menu items for your own custom commands

The debugger stores information about customized buttons and menu items in the file **.clc++usrcmd**. The debugger automatically generates this file and saves it in your home directory at the end of your session. Although **.clc++usrcmd** is an editable ASCII file, we recommend that you do not edit it.

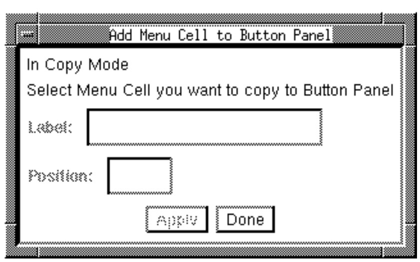
Adding new buttons

You can add a button for any menu command already available in the Main Window. To add a button for any other command, such as a Workspace command, you need to create a custom command first, as described in “Creating new menu items for custom commands” on page 195. Once a menu command is available, follow these steps to create a button for it:

1 Display the **CenterLine-C++** menu and then the **Button Panel** submenu.

2 Select **Add Menu Items to Panel**.

The debugger opens the Add Menu Cell to Button Panel dialog and places the Main Window in copy mode, as shown in the following illustration:



3 Display the desired menu in the menu bar and select the menu item for which you want to create a button.

NOTE You cannot select a menu item that appears on a submenu.

The **Label** text field displays the name of the button, and the **Position** text field displays the position, which defaults to 0 (the leftmost button in the button panel).

4 If desired, edit the **Label** and **Position** text fields.

5 Select the **Apply** button.

The new button appears at the specified position on the Button panel.

6 When finished, select the **Done** button.

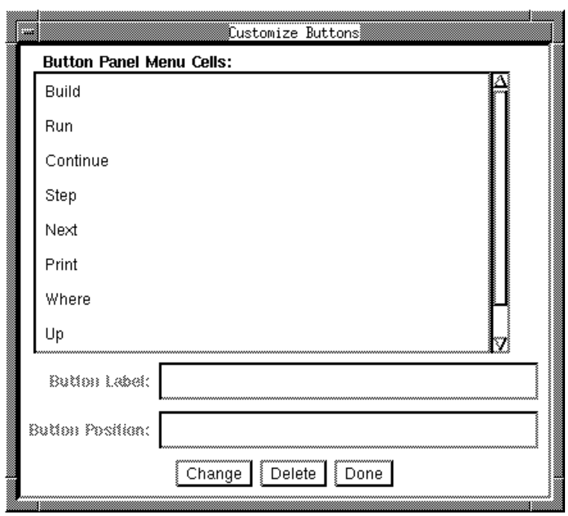
Changing the name and position of buttons

To change the name and position of a button:

1 Display the **CenterLine-C++** menu and then the **Button Panel** submenu.

2 Select **Customize Button Panel**.

The debugger opens the Customize Buttons dialog, as shown in the following illustration.



3 Select the desired button in the scrolling list.

4 Edit the **Button Label** and **Button Position** text fields.

5 Select the **Change** button.

6 When finished, select the **Done** button.

Deleting buttons

To delete a button:

1 Display the **CenterLine-C++** menu and then the **Button Panel** submenu.

2 Select **Customize Button Panel**.

The debugger opens the Modify Buttons in Button Panel dialog.

3 Select the desired button in the scrolling list.

4 Select the **Delete** button.

5 When finished, select the **Done** button.

Creating new menu items for custom commands

You can create custom commands that appear on the **User Defined** submenu of the **CenterLine-C++** menu.

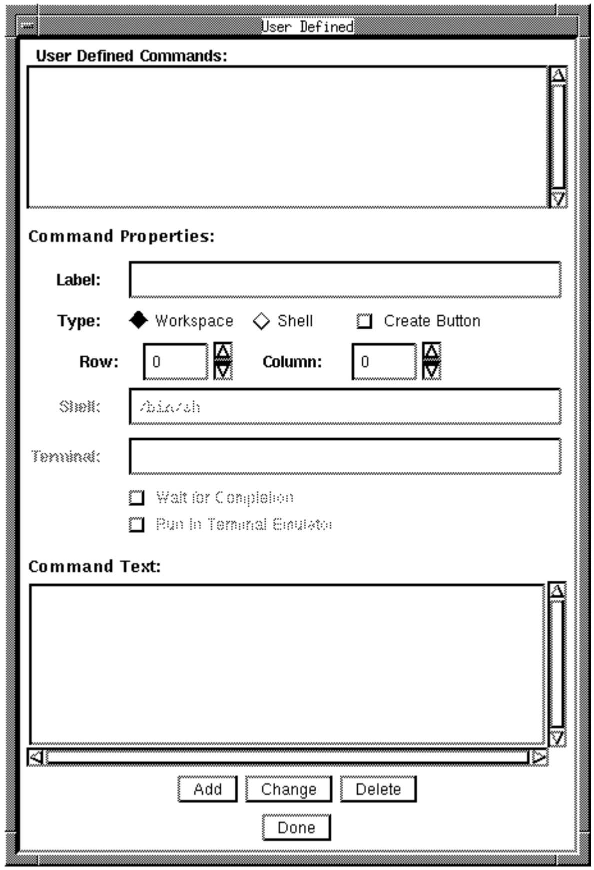
1 Display the **CenterLine-C++** menu and then the **User Defined** submenu.

2 Select **Add/Change/Delete**.

The debugger opens the User Defined dialog, as shown in the illustration on the next page. .

3 Enter the name of the custom command in the **Label** text field.

4 If you want to add a button to the button panel for this command, select the **Create Button** button. You also need to specify the row and column position of the button if the default (row 0, column 0) isn't suitable.



5 Decide whether your command will use Workspace commands (default) or shell commands.

If the latter, select the **Shell** button. You also need to specify the following items.

- The shell you want the debugger to fork when you invoke this custom command.
- Whether you want the debugger to wait for all the shell commands in the definition to terminate before continuing its own process.
- Whether you want the shell output to use a terminal emulator. If so, you specify which one to use. If you do choose *not* to use a terminal emulator, your commands will not be able to perform any input or output. Use this choice only if your commands do not do any input and if you do not want to see any command output.

6 Enter the definition for the command in the **Definition** text box. You can enter as many commands as you like.

For custom Workspace commands, on each line in the Definition area, put any input that the Workspace will accept on a single line. You cannot use a backslash (\) to escape the newline character.

For custom shell commands, put any input that the specified shell will accept.

When you define a custom command, you can use the following variables:

$pwd	The debugger's current working directory.
$filename	The filename of the file in the Source area, relative to the debugger's current working directory.
$filepath	The absolute filename of the file in the Source area.
$selection	The X11 *PRIMARY* selection, interpreted as a string. The X11 PRIMARY selection is the range of text that you drag to select. If the selection is not available or is empty, **$selection** is replaced with an empty string.
$clipboard	The X11 *CLIPBOARD* selection. The X11 CLIPBOARD selection is the range of text that you drag to select and then press the Copy key or use the Copy menu item (usually in OPEN LOOK only). If the selection is not available or is empty, **$clipboard** is replaced with an empty string.
$first_selected_line	The line number (numbered from 1 to *n*) of the first line of the current text selection in the Source area of the Main Window.

$first_selected_char	The character position (numbered from 1 to *n*, with tabs being a single character) of the first character that is selected in **$first_selected_line**.
$last_selected_line	The line number (numbered from 1 to *n*) of the last line of the current text selection in the Source area of the Main Window.
$last_selected_char	The character position (numbered from 1 to *n*, with tabs being a single character) of the last character that is selected in **$last_selected_line**.

If there is no text selection in the Source area, the **$first_selected_line**, **$first_selected_char**, **$last_selected_line**, and **$last_selected_char** return 0.

NOTE If the current X11 selection contains newlines, the **$clipboard** and **$selection** variables expand to multiple lines. You cannot use multiple lines for customized Workspace commands. Multiple lines might also interfere with customized shell commands.

Modifying custom commands

To modify a custom command:

1. Display the **CenterLine-C++** menu and then the **User Defined** submenu.
2. Select **Add/Change/Delete**.

   The debugger opens the User Defined dialog.
3. Select the command from the scrolling list.
4. Make the changes desired.
5. Select the **Change** button.
6. When finished, select the **Done** button.

Deleting custom commands

To delete a custom command:

1. Display the **CenterLine-C++** menu and then the **User Defined** submenu.
2. Select **Add/Change/Delete**.

   The debugger opens the User Defined dialog.
3. Select the command from the scrolling list.
4. Select the **Delete** button.
5. When finished, select the **Done** button.

Example

To add a button for the Quit menu command:

1. Display the **CenterLine-C++** menu and then the **Button Panel** submenu.
2. Select **Add Menu Items to Panel**.

   The debugger opens the Add Menu Cell to Button Panel dialog and places the Main Window in copy mode.
3. Display the **CenterLine-C++** menu and select the **Quit CenterLine-C++** menu item.

   The Label text field displays the name of the button, and the Position text field displays the position, which defaults to 0 (the leftmost button in the button panel).
4. Change the button label in the Label field to **Quit**.
5. Select the **Apply** button.
6. Select the **Done** button to close the dialog.
7. Select the **Quit** button in the Button panel to test the new button.

See also

X resources in the Man Browser

Task: Customizing environment variables

To display, set, and unset the environment variables that are used by *your program*, you can use the **printenv**, **setenv**, and **unsetenv** commands in the debugger. They are analogous to the similarly named **csh** commands.

These commands affect only your program's environment variables; they do not affect the environment variables used by the debugger.

printenv	Displays the values of environment variables.
setenv	Sets the value of an environment variable.
unsetenv	Unsets an environment variable.

printenv displays the default values of the environment variables, which are the values that your program inherits each time it starts. If your program alters an environment variable with the **putenv()** library function, the change is not shown by the **printenv** command.

For example, you can display and set the current setting for the **SHELL** environment variable in the following way:

```
pdm -> printenv SHELL
SHELL=/usr/local/bin/tcsh
pdm -> setenv SHELL /bin/sh
pdm -> printenv SHELL
SHELL=/bin/sh
```

See also

printenv on page 236

setenv on page 243

unsetenv on page 256

Task: Quitting from the debugger

You can exit from the debugger using one of the following methods:

- Display the **CenterLine-C++** menu in the Main Window and select **Quit.** In the dialog box, select **Quit** to exit.
- Enter the **quit** command in the Workspace.

7 Command Reference

This chapter contains an alphabetical reference to commands available in pdm, the CenterLine-C++ symbolic debugger.

alias

creates an alias for a command

Command syntax

alias
alias *name*
alias *name text*

Description

<< *none* >>	Lists all aliases currently set.
name	Lists the text value for the specified alias *name*.
name text	Sets the *name* string to the value of the *text* string.

Usage

Use the **alias** command to create an alternative name for CenterLine-C++ commands. When an alias is detected at the beginning of a command line, its text is used in place of the name. Use aliases to create shortcuts for frequently used commands.

Default aliases

In addition to aliases that you can create, CenterLine-C++ comes with several default aliases, such as **ls** and **pwd**. When you issue the **alias** command without arguments, the default aliases are displayed along with any that you have defined. For example:

```
-> alias s step
-> alias
ls                  sh ls
pwd                 sh pwd
s                   step
```

NOTE To save an alias permanently, place its definition in your **.pdminit** file.

See also

unalias

"Using aliases for Workspace commands" on page 136

"Customizing your startup file" on page 191

assign

assigns a value to a variable

Command syntax

assign *variable = expression*

Description

variable = expression	Evaluates an expression (second argument) and assigns the value of the expression to a variable (first argument).

Usage

Use the **assign** command to evaluate an expression and assign its value to a variable. Assigning a value to a variable in the Workspace allows you to either directly manipulate values in code that you are debugging or to set values for code you are creating in the Workspace. The **assign** and **set** commands are functionally identical.

See also

print, set

"Examining data structures" on page 169

attach

attaches to a running process

Command syntax

attach *process_id*

Description

process_id	Attaches CenterLine-C++ to the running process identified by *process_id*. You can attach to only one process at a time.

Usage

When you attach to a running process, CenterLine-C++ stops the process. You can then examine and modify the process with any CenterLine-C++ command. If you want the process to continue running, use the **cont** command. Use the **detach** command to release a process from CenterLine-C++'s control.

If you try to attach a process while you are already attached to another process, CenterLine-C++ prompts you to detach before attaching.

You can use the **attach** command in combination with **debug** to attach an executable file to an already running process. That is, you can use the following two commands:

```
(pdm) 1 -> debug my_a.out
(pdm) 2 -> attach my_process_id
```

instead of the following:

```
(pdm) 2 -> debug my_a.out my_process_id
```

NOTE If you use the **run** command while you have an attached process, you kill that process.

See also

debug, detach

"Debugging a running process" on page 182

build

reloads the program currently being debugged if it is out of date

Command syntax **build**

Description

<< *none* >>	Reloads the executable (for instance, **a.out**) if the executable is newer than the current one.

Usage

Use the **build** command to reload the file that you are debugging after you have recompiled it.

See also

debug, make

“Building and reloading executables” on page 147

catch

traps signals before they reach the program

Command syntax

catch
catch *signal_name*
catch *signal_number*

Description

<< *none* >>	Lists the unprefixed names of the signals that are currently caught.
signal-name	Enables trapping for the designated signal and generates a break level whenever the signal is generated.
signal-number	Enables trapping for the designated signal and generates a break level whenever the signal is generated.

Usage

Use the **catch** command to trap signals before they reach the program; each signal is either caught or ignored by CenterLine-C++. Once a signal is trapped, CenterLine-C++ generates a break level.

You can use the **cont** command to pass the signal number to your program.

Signal numbers

To obtain the number for a signal, consult the UNIX reference manuals for your system.

Signals caught

To view a list of the signals caught for your platform, use the **catch** command without any arguments.

Signal name

With the **catch** command, the signal name can be in uppercase or lowercase letters, and it can be used with or without the prefix “SIG”. For example, the following commands are equivalent:

```
-> catch SIGALRM
-> catch sigalrm
-> catch ALRM
-> catch alrm
```

Restrictions

Control-z during execution or in the Run Window is always handled as a signal-deliver, generating an error if not trapped by the user program.

Ignoring SIGINT causes SIGQUIT to perform interruption duties. Ignoring both of them interferes with stopping execution.

The signals SIGTTIN and SIGTTOU will never suspend execution; if not trapped and ignored they will generate an error.

See also

cont, ignore

“Handling signals” on page 177

cd

changes the current working directory

Command syntax

cd
cd *pathname*

Description

<< *none* >>	Changes the working directory for CenterLine-C++ to your home directory.
pathname	Changes the working directory for CenterLine-C++ to the designated pathname. UNIX wildcards are allowed.

Usage

To facilitate loading and saving files, use the **cd** command to change the current working directory for CenterLine-C++.

See also

use

cont

continues execution from a break level

Command syntax

cont

cont at *line*

cont at *line* **sig** *signum*

cont sig *signum*

cont skip *count*

Description

<< *none* >>	Continues execution of the program from the current break level.
at *line*	Continues at location specified by *line*.
at *line* **sig** *signum*	Continues at location specified by *line* with signal specified by *signum*. This means the signal is delivered to your program, which must handle it.
sig *signum*	Continues with signal specified by *signum*.
skip *count*	Continues, ignoring breakpoint for *count* iterations.

Usage

Use the **cont** command to continue execution of the program from a break level.

Restrictions

You cannot continue from all errors by supplying a continuation value to **cont**.

See also

step, stepout, stop, where, whereami

“Using Workspace break levels” on page 160

“Running, continuing, and stepping” on page 163

contents

lists source files for the current debug file

Command syntax

contents
contents all
contents *file*

Description

<< *none* >>	Returns the pathname of the **a.out** file currently loaded.
all	Lists known source files for the **a.out** file currently being debugged.
file	Lists the functions defined in the source file named *file.*

Usage

Use the **contents** command to display information about files in the executable that you are currently debugging. The **contents** command lists only the files that were compiled with debugging information (the **-g** switch).

Restrictions

The following restrictions apply:

- The **contents all** variation does not display files compiled without debugging information.
- The **contents** *file* variation may return only a partial list of objects declared or defined in *file.*

See also

build, make

"Listing source code" on page 143

debug

loads an executable file, a corefile, or a process for debugging

Command syntax

debug

debug *executable*

debug *executable corefile*

debug *executable process_id*

Description

<< *none* >>	Displays the name and arguments of the program being debugged.
executable	Loads the symbol table for *executable*, which is the name of the executable program to be debugged.
executable corefile	Loads the symbol table from the executable program (*executable*) and sets up CenterLine-C++ to work with the *corefile* along with the executable program. The *corefile* contains a literal copy of the contents of memory at the time that the operating system aborted a program.
executable process_id	Loads the symbol table from the *executable* file and attaches to the running process identified by *process_id*. The process can be running inside or outside of CenterLine-C++.

Usage

Use the **debug** command to load the files required for the following kinds of source-level and machine-level debugging:

- Debugging a fully linked executable program
- Debugging a fully linked executable program along with a corefile
- Debugging a running process

Using the -g switch

The information in the symbol table in an *executable* file varies according to whether or not you used the **-g** switch when you compiled the object modules that you linked to create it. Modules that are not compiled with **-g** contain the information for machine-level debugging only, plus information about the hexadecimal address of external symbols. Modules compiled with **-g**, in contrast, contain full source-level debugging information. Also, if you strip debugging information from an *executable* file, you are limited to machine-level debugging without any knowledge of external symbols.

Using run after debug

After you use **debug** to load a program, use **run** to start it running. This causes CenterLine-C++ to create a process and make that process run your program. You can then use any CenterLine-C++ command to debug the program.

If your program crashes and creates a corefile, you can use **debug** to load the corefile created when it crashed.

Attaching to a process

When you attach to a running process, the first thing that CenterLine-C++ does is to stop the process. You can then examine and modify the process with the commands available in CenterLine-C++. If you want the process to continue running, use the **continue** (**cont**) command. Use the **detach** command to release a process from CenterLine-C++'s control.

You can use the **attach** command in combination with **debug** to attach to an already running process. That is, you can use the following two commands:

```
pdm  1 -> debug my_executable
pdm  2 -> attach my_process_id
```

instead of the following:

```
pdm 3 -> debug my_executable my_process_id
```

NOTE If you use the **run** command while you have an attached process, you kill that process.

See also

attach, detach

"Starting up the debugger" on page 120

"Loading files for debugging" on page 142

"Debugging an executable with a corefile" on page 178

"Debugging a running process" on page 182

delete

deletes debugging items

Command syntax

delete **all**
delete *number* ...

Description

all	Deletes all debugging items everywhere.
number...	Deletes the specified debugging item.

Usage

Use the **delete** command to delete a breakpoint, action, or display.

To obtain the number of a debugging item, use the **status** command.

Zombied items

If **delete** is called on a debugging item currently active on the execution stack, the item will be **zombied** (marked for deletion) instead of being deleted immediately. A zombied item is deleted once it has completed executing.

See also

display, status, stop, when

"Setting breakpoints and watchpoints" on page 150

"Setting actions" on page 155

"Examining and deleting debugging items" on page 158

"Using Workspace break levels" on page 160

detach

detaches from a running process

Command syntax **detach**

Description

<<*none*>>	Detaches CenterLine-C++ from the running process that was attached using CenterLine-C++'s **attach** command.

Usage

Use the **detach** command to release a process from CenterLine-C++'s control. Detaching a process continues its execution.

After you use the **detach** command, a process is completely independent of CenterLine-C++, and you can use **attach** with another process, or start a process with **run**.

NOTE If you use the **run** command while you have an attached process, you kill that process.

See also

attach, debug

"Debugging a running process" on page 182

display

displays the value of a variable or expression

Command syntax

display *expression*
display *variable*

Description

expression	Invokes the Data Browser, which creates a new display item each time you invoke the **display** command. The display item graphically displays the value of the variable or expression.
variable	Displays the value of the designated global or local variable whenever execution is stopped.

Usage

Use the **display** command to display the value of an expression or a variable. CenterLine-C++ displays the value whenever your program is stopped, including during single-stepping.

Local variables

The argument to **display** may contain references to local variables that are currently in scope. If execution later stops at a point where these variables are no longer in scope, the **display** will either generate an error (Workspace) or show the variable with the text in a dimmed or "greyed out" state (Motif or OPEN LOOK).

Manipulating display items

Display items can be deleted with the **delete** command and examined with the **status** command.

See also

delete, dump, print, status, whatis, whereis

“Examining and deleting debugging items” on page 158

“Examining data structures” on page 169

down

moves down the execution stack

Command syntax	**down** **down** *number*

Description

<< *none* >>	Moves the current scope location down one level on the execution stack. Source panel shows file scoped to location and highlights it with an arrow.
number	Moves the current scope location the specified number of levels down on the execution stack.

Usage

Use the **down** command to move the current scope location down the execution stack, away from the top level of the Workspace and toward the current break level.

The scope location is the point at which all variables, types, and macros are scoped. When a break level is generated, the scope location is set to the point at which execution was interrupted.

When at a break level, the **where** command can be used to display the execution stack. The **whereami** command can be used to display the break location and the current scope location.

See also

cont, reset, up, where, whereami

"Using Workspace break levels" on page 160

"Moving in the execution stack" on page 166

dump

displays all local variables

Command syntax

dump
dump *function*
dump *text*

Description

<< *none* >>	Displays the name and value of each variable local to the current scope location.
function	Displays the name and value of each variable local to the specified function.
text	Displays the name and value of each variable contained in an arbitrary text string.

Usage

Use the **dump c**ommand to display the names and values of local variables.

Typically you use the *text* argument by selecting a range of text, then issuing the **dump** command. CenterLine-C++ displays the name and value of each of the variables contained in the text string.

See also

display, print, whatis, whereis

"Using Workspace break levels" on page 160

"Examining data structures" on page 169

edit

invokes your editor at a specified location

Command syntax

edit
edit *file*
edit *"file":line*
edit *function*
edit *line number*

Description

<< *none* >>	Loads the current file into your editor, positioned at the current list location.
file	Loads the specified file into your editor, positioned at the top of the file.
"file":line	Loads the specified file into your editor, positioned at the specified line in the file.
function	Loads the file containing the specified function definition into your editor, positioned at the start of the function.
line number	Loads the file specified by the current list location into your editor, positioned at the specified line number.

Usage

Use the **edit** command to facilitate quick debug-edit-run turnaround times by invoking your editor to edit a file at a specified location. In all cases, once the editor is invoked, the current list location is set to the file and line number edited.

See also

“Selecting an editor to use with the debugger” on page 119

“Using menus and text fields” on page 126

“Copying and pasting text between windows” on page 129

“Editing Workspace input” on page 137

“Editing source code” on page 145

email

sends electronic mail to CenterLine Software

Command syntax **email**

Description

<< *none* >>	Opens the **email** dialog box.

Usage

To report bugs or offer suggestions, use the **email** command to send an electronic mail message to CenterLine Software. When you send a bug report, include examples of the source code that produced the problem, if possible.

When you issue the **email** command, you can use the UNIX **mail(1)** electronic mail utility's escape sequences.

See also

"Finding out more about the debugger" on page 122

file

displays and sets the current list location

Command syntax

file
file *filename*

Description

<< *none* >>	Displays the name of the file containing the current list location.
filename	Sets the current list location to the top of the specified file.

Usage

Use the **file** command to display and set the current list location. Commands such as **edit**, **list**, and **stop** use the list location as the default location unless specifically overridden by an argument.

The **file** command changes which static variables are visible at the top level in the Workspace.

See also

edit, list, stop

"Using Workspace break levels" on page 160

gdb

executes a **gdb** command

Command syntax

gdb *gdb_command* **[** *argument* **]** ...

Description

gdb_command [*argument*]	Executes *gdb_command* [*argument*] as if it were typed to a **gdb** command prompt.

Usage

The **gdb** command allows you to stay in CenterLine-C++ and execute **gdb** commands. For instance, the following invokes **break**, a **gdb** command, with **20** as the argument:

```
pdm 1 -> gdb break 20
```

NOTE Although we provide access to native **gdb** commands as a convenience, we do not provide any additional support for native **gdb** commands**.**

For more information on **gdb** commands, you can use the **gdb help** command:

```
pdm 1 -> gdb help
```

Documentation on **gdb** is available from CenterLine by using anonymous **ftp**. For information, refer to “Distribution” on page iii.

See also

gdb_mode

“Invoking Workspace commands” on page 131

gdb_mode

invokes **gdb**

Command syntax

gdb_mode

Description

<< *none* >> Invokes **gdb**.

Usage

Use the **gdb_mode** command when you want to issue a series of **gdb** commands without prefacing every command with the **gdb** command.

To use **gdb** along with **pdm**, issue the **gdb_mode** command in the CenterLine-C++ Workspace:

```
pdm 1 -> gdb_mode
(gdb)
```

At the **(gdb)** prompt, you can use *only* the **gdb** command set:

```
(gdb) break 20
(gdb) when
Undefined command: "when". Try ”help”.
```

You can get back to the **pdm** debugger by typing the following command:

```
(gdb) pdm
pdm 2 ->
```

NOTE Although we provide the **gdb_mode** command as a convenience, we do not provide any technical support for **gdb.**

For more information on **gdb** commands, you can use the **help** command while in **gdb** mode.

Documentation on **gdb** is available from CenterLine by using anonymous **ftp**. For information, refer to “Distribution” on page iii.

See also

gdb

“Invoking Workspace commands” on page 131

help

displays usage information about commands

Command syntax

help
help *command*

Description

<< *none* >>	Lists the names of CenterLine-C++ commands by category.
command	Displays a summary of syntax and usage information for the specified command.

Usage

Use the **help** command for quick online help for CenterLine-C++ commands.

See also

man

"Finding out more about the debugger" on page 122

history

lists previously entered input

Command syntax

history
history *number*

Description

<< *none* >>	Displays all input lines previously entered from the Workspace.
number	Displays the specified number of input lines entered from the Workspace.

Usage

Use the **history** command for easy recall of previously issued commands and to monitor the debugging sequence leading to a given state.

Use **##** to repeat the immediately previous command, and use **#***history_line_number* to repeat the command specified by *history_line_number.*

Pressing Control-p scrolls backward through the history list. Pressing Control-n scrolls forward through the history list.

To save the list of input lines entered from the Workspace in a file, use the following command:

```
-> history #> file_name
```

See also

"Saving your debugging session" on page 190

ignore

allows signals to pass directly to the program

Command syntax

ignore
ignore *signal-name*
ignore *signal-number*

Description

<< *none* >>	Lists the unprefixed name of the signals that are currently ignored.
signal-name	Disables trapping for the designated signal, allowing the signal to pass directly to the program, which can execute a signal handler if it has been specified.
signal-number	Disables trapping for the designated signal, allowing the signal to pass directly to the program, which can execute a signal handler if it has been specified.

Usage

Use the **ignore** command for any signal that you want to pass directly to the program. Once an ignored signal is passed to the program, the program executes any signal handlers specified for it.

Signal numbers

To obtain the number for a signal, consult the UNIX reference manuals for your system.

Signal names

With the **ignore** command, the signal name can be in uppercase or lowercase letters, and it can be used with or without the prefix “SIG”. For example, the following commands are equivalent:

```
-> ignore SIGHUP
-> ignore sighup
-> ignore HUP
-> ignore hup
```

Signals ignored

To obtain a list of signals ignored on your platform, type the **ignore** command without any arguments.

NOTE Even if a signal is ignored, it interrupts system calls, such as **select()**, that are interruptible.

Restrictions

When a signal is caught and a break level is generated, the signal is consumed. Ignoring the signal at the break level and continuing execution will not regenerate the signal and pass it to the program.

Control-z during execution or in the run window is always handled as a signal-deliver, generating an error if not trapped by the user program.

Ignoring SIGINT causes SIGQUIT to perform interruption duties. Ignoring both of them interferes with stopping execution.

The signals SIGTTIN and SIGTTOU will never suspend execution; if not trapped and ignored they will generate an error.

See also

catch

"Handling signals" on page 177

list

displays source code lines

Command syntax

list
list *file*
list *"file":line*
list *function*
list *line_number*
list *start_line end_line*

Description

<< *none* >>	Lists source code starting at the current list location.
file	Lists source code starting at the top of the specified file.
"file":line	Lists source code starting at the specified line number in the specified file.
function	Lists source code starting at the top of the specified function.
line_number	Lists source code starting at the line number specified.
start_line end_line	Lists source code starting at the line number specified by *start_line* and ending at the line number specified by *end_line*.

Usage

Use the **list** command to display specific lines of source code relative to the current **list location**. The list location is set by the following events:

- When a file is loaded, it is set to the first line.
- When a break level is entered, it is set to the break location.
- When the **list** command is used, it is set to the last line displayed.

You can also set the list location using the **file** command.

If you use the **list** command and specify a static function for the *function* argument, you may receive an error in certain situations. However, if you first use the **whatis** command and specify the static function as an argument, the debugger loads additional symbols. Then, you can use the **list** command with the static function to load the source code in the Source area.

See also

display, edit, whatis, whereis

“Listing source code” on page 143

listi

displays machine instructions

Command syntax

listi

listi *addr*

listi *addr1 addr2*

listi *line*

listi *line1 line2*

listi *func*

listi *func + offset*

Description

<<*none*>>	Displays machine instructions at current program counter address.
addr	Displays machine instructions at *addr*. The value of *addr* can be a hexadecimal or octal number.
addr1 addr2	Displays machine instructions between *addr1* and *addr2*. The values of *addr1* and *addr2* can be hexadecimal or octal numbers.
line	Displays machine instructions at *line* in current file. The value of *line* must be a decimal number.
line1 line2	Displays machine instructions between *line1* and *line2*. The values of *line1* and *line2* must be decimal numbers.
func	Displays machine instructions for *func*.
func + offset	Displays machine instruction at the address equal to the address of *func* plus *offset*.

See also

list, nexti, stepi, stopi

“Listing source code” on page 143

“Debugging machine instructions” on page 186

make

invokes the UNIX **make** command

Command syntax

make
make *target ...*

Description

<< *none* >>	Calls the UNIX **make** command using the default target. Shows **make** errors in the Error Browser.
target ...	Calls the UNIX **make** command using the *target* argument as its target.

Usage

The CenterLine-C++ **make** command has the same effect as using the **make** command in the shell. The Error Browser displays any **make** errors.

See also

build, contents

"Building and reloading executables" on page 147

"Finding and fixing errors" on page 148

man

displays information about CenterLine-C++ commands

Command syntax

man
man *command*

Description

<< *none* >>	Invokes the Man Browser.
command	Invokes the Man Browser, and opens the entry for the specified command.

Usage

Use the **man** command to get online information for CenterLine-C++ commands.

See also

help

"Finding out more about the debugger" on page 122

next

executes source code by line; does not enter functions

Command syntax

next
next *number*

Description

<< *none* >>	Executes an entire line, regardless of the number of statements on the line, and then stops execution. Displays a solid arrow pointing to the current execution line in the Source area.
number	Executes the specified number of lines, and then stops execution.

Usage

Use the **next** command to execute your code line by line without going into functions that are called.

The **next** command does not stop inside object code functions that do not have debugging information (functions compiled without the **-g** switch).

See also

nexti, step, stepout

“Using Workspace break levels” on page 160

“Running, continuing, and stepping” on page 163

nexti

executes machine code by line; does not enter functions

Command syntax

nexti
nexti *num*

Description

<<none>>	Executes the next line of machine code, but does not enter functions.
num	Executes *num* machine instructions, not just the last one, but does not enter functions.

See also

listi, next, stepi, stopi

"Using Workspace break levels" on page 160

"Running, continuing, and stepping" on page 163

"Debugging machine instructions" on page 186

print

prints the value of variables and expressions

Command syntax

print *expression*
print *variable*

Description

expression	Evaluates the specified expression and displays the resulting value.
variable	Displays the value of the specified variable.

Usage

Use the **print** command to check the current value of variables and expressions. CenterLine-C++ prints their values in the Workspace.

```
-> print r
(struct Rectangle *) 0x8a98
```

The value of a variable or expression can also be displayed without the **print** command. This is accomplished by evaluating the variable or expression directly in the Workspace:

```
-> print 123+456
(long) 579
-> 123+456
(long) 579
->
```

See also

assign, display, dump, list, whatis, whereis

"Using Workspace break levels" on page 160

"Examining data structures" on page 169

printenv

displays the system environment

Command syntax

printenv
printenv *variable*

Description

<< *none* >>	Lists all currently defined environment variables.
variable	Displays the value of the specified environment variable, if that variable is currently defined.

Usage

Use the **printenv** command in conjunction with **setenv** and **unsetenv** to manipulate the variables in the program's system environment. The **printenv** command is similar to the shell command with the same name.

Warnings

The **printenv** command displays the default values of the environment variables, which are the values that your program inherits each time it starts. Therefore, if a program has added any environment variables, for instance with the **putenv()** library function, the changes will not be shown by the **printenv** command.

If **setenv** or **unsetenv** is called from a break level, they will alter the value of the global **environ** variable, but not the **envp** parameter passed to **main()**. (This problem also occurs with the **putenv()** function.)

Changing the **EDITOR** or **DISPLAY** shell variables with these commands will not affect which editor or display screen CenterLine-C++ uses.

See also

setenv, unsetenv

"Customizing environment variables" on page 200

quit

quits CenterLine-C++

Command syntax

quit

Description

<< *none* >>	Exits CenterLine-C++ and returns you to the shell.

Usage

Use the **quit** command to exit CenterLine-C++ and return to the shell. Before exiting, CenterLine-C++ notifies you if there are any active editing jobs.

See also

“Quitting from the debugger” on page 201

rerun

executes **main()** with new arguments

Command syntax

rerun
rerun *argument* ...

Description

<< *none* >>	Clears any old command-line arguments, initializes all variables, and then executes **main().**
argument ...	Clears any old command-line arguments, initializes all variables, processes the new command-line arguments (*argument* ...), and then executes **main()**. If you issue a **rerun** command while you are at a breakpoint, CenterLine-C++ restarts and prompts you before resetting the break level.

Usage

Use the **rerun** command to execute **main()** with new arguments.

Arguments must be delimited by spaces. To include spaces in an argument string, precede each space with a backslash (\) character. Calling a program with **rerun** produces the same results as calling an executable program from the shell.

Restrictions

In the Workspace, to pass an argument with a space in it to **main(),** you must escape it with a backslash. Enclosing the argument in quotation marks, which works in a UNIX shell, does not work in the Workspace. For example, to call **main()** with two arguments, the first one containing the string **first arg**, and the second argument containing the number **3**, call **rerun** as follows:

```
-> rerun first\ arg 3
```

In contrast, you can use double quotes just as you do in a UNIX shell when you supply arguments for the **Run** dialog box.

See also

run

“Running, continuing, and stepping” on page 163

reset

returns to a previous break level

Command syntax **reset**

Description

<< *none* >>	Returns execution to the top level of the Workspace.

Usage

Use the **reset** command to return to a previous break level without continuing execution from the current break level. When you issue the **reset** command the executable is killed and its resources are freed.

See also

cont, stop, where, whereami

"Running, continuing, and stepping" on page 163

run

executes **main()** with arguments

Command syntax

run
run *argument* ...

Description

<< *none* >>	Initializes all variables, processes any command-line arguments from the previous call to **run** or **rerun**, and then executes **main()**.
argument ...	Clears any old command-line arguments, initializes all variables, processes the new command-line arguments (*argument* ...), and then executes **main()**.

When you run your program in CenterLine-C++, its output goes to the Run Window.

Any arguments that you supply with the **run** command are first passed to a shell, which expands wildcard characters, substitutes variables, and redirects I/O, and then passed to **main()**. The value of the SHELL environment variable, as outlined in Table 12, specifies the shell to be used for processing these arguments..

Table 12 Shells Used with the **run** Command

Value of SHELL Environment Variable	What CenterLine-C++ Does
No SHELL environment variable	Uses **/bin/sh**
/bin/tcsh	Uses **/bin/csh** instead of **/bin/tcsh**[a]
/bin/csh	Uses **/bin/csh** with the **-f** flag[b]
All other values not listed	Invokes shell with the **-c** option.

a. This avoids a problem with **tcsh,** where the first file descriptor that the user program gets is 6 instead of 3.

b. This keeps the shell from reading your startup file and improves speed.

Usage

Use the **run** command to execute **main()** after initializing all variables and processing any command-line arguments.

If you issue a **run** command while you are at a breakpoint, CenterLine-C++ restarts and informs you that it is resetting the break level.

How CenterLine-C++ interprets the command

Both **run** and **rerun** construct arguments for **main()** from the command line. If **run** is called without any arguments, it uses the command-line arguments from the most recent call to either **run** or **rerun**. If **rerun** is called without any arguments, it calls **main()** without any arguments.

Passing arguments containing spaces

In the Workspace, in order to pass an argument with a space in it to **main(),** you must precede the space with a backslash. Enclosing the argument in quotation marks, which works in a UNIX shell, does not work in CenterLine-C++.

For example, to call **main()** with two arguments, the first one containing the string **first arg**, and the second argument containing the number **3**, call **run** as follows:

```
-> run first\ arg 3
```

In contrast, you can use quotation marks just as you do in a UNIX shell when you supply arguments for the **Run** dialog box.

See also

rerun

“Running, continuing, and stepping” on page 163

set

assigns a value to a variable

Command syntax

set *variable* **=** *expression*

Description

variable **=** *expression*	Evaluates *expression* and assigns its value to *variable*.

Usage

Use the **set** command to assign a value to a variable. The specified variable can be a variable defined in either the program or the Workspace.

See also

assign

“Examining data structures” on page 169

setenv

adds a variable to the system environment

Command syntax

setenv
setenv *variable*
setenv *variable value*

Description

<< *none* >>	Lists all defined environment variables and gives their current values. This is equivalent to calling **printenv** without an argument.
variable	Defines *variable* and sets its value to the empty string. If the specified variable already exists, its value is reset to the empty string.
variable value	Defines *variable* and sets it to the value specified by *value*. If *variable* already exists, its value is reset to *value*.

Usage

Use the **setenv** command to manipulate the variables in the program's system environment. The **setenv** command is analogous to the shell command of the same name.

These commands affect only your program's environment variables. They do not affect the environment variables used by CenterLine-C++ to control its own operations.

The environment is an array of strings that is made available to the program through the global **environ** variable and the **envp** parameter, which is passed as the third argument to the **main()** function. By convention, each string has the format **name=value**, where the **value** part is optional.

Warnings

Be careful when checking the current values for environment variables. The **printenv** and **setenv** commands, when issued with no argument, display the default values of the environment variables, which are the values that your program will inherit each time it starts.

If **setenv** or **unsetenv** are called from a break level, they will alter the value of the global **environ** variable, but not the **envp** parameter passed to **main()**. This problem also occurs with the **putenv()** function.

Changing the **EDITOR** or **DISPLAY** shell variables with these commands will not affect which editor or display screen CenterLine-C++ uses.

See also

printenv, unsetenv

"Customizing environment variables" on page 200

sh

executes a Bourne subshell

Command syntax

sh
sh *argument* ...

Description

<< *none* >>	Executes a Bourne subshell, setting no switches and passing no arguments.
argument ...	Executes a Bourne subshell, setting the **-c** switch and passing the specified arguments.

Usage

Use the **sh** command to execute a Bourne subshell. This can be used to execute UNIX commands from the Workspace:

```
-> sh rm my_file
```

See also

shell

“Invoking shell commands” on page 135

shell

executes a subshell

Command syntax

shell
shell *argument ...*

Description

<< *none* >>	Executes the default shell specified by the SHELL environment variable. Sets no switches and passes no arguments.
argument ...	Executes the default shell specified by the SHELL environment variable. Sets the **-c** switch and passes *argument* to the shell. You can have more than one argument.

Usage

Use the **shell** command to execute the shell specified by the **shell** option. This can be used to execute UNIX commands in the Workspace.

See also

sh

"Invoking shell commands" on page 135

source

reads CenterLine-C++ commands from a file

Command syntax

source *file*

Description

file	Reads CenterLine-C++ commands from the specified file.

Usage

Use the **source** command to read CenterLine-C++ commands from a file.

CenterLine-C++ uses **source** to read the system-wide startup file and the **.pdminit** file in your home or current directory when you start CenterLine-C++.

Example

The following example indicates how to use **source** with a file containing aliases:

```
% cat aliases
alias p print
alias s step
alias n next
alias ls sh ls
% centerline-c++
.
.
.
-> source aliases
-> p 123+456
(long) 579
->
```

status

lists debugging items (actions, breakpoints, and displayed items)

Command syntax

status

Description

<< *none* >>	Lists all currently set debugging items.

Usage

Use the **status** command to list all breakpoints, actions, and displays. This listing displays the debugging item number needed for the **delete** command.

Zombied items

If the **delete** command has been invoked on a debugging item that is currently active on the execution stack, **status** reports the item as **zombied**. When execution continues, the zombied item will be deleted once it has completed executing, and **status** will no longer list it.

See also

delete, display, stop, when

“Setting breakpoints and watchpoints” on page 150

“Setting actions” on page 155

“Examining and deleting debugging items” on page 158

“Examining data structures” on page 169

step

steps execution by statement, entering functions

Command syntax

step
step *number*

Description

<< *none* >>	Executes a single statement and then stops execution. Updates the Source area to display the new line of execution.
number	Executes the specified number of statements and then stops execution.

Usage

Use the **step** command to single-step through your program, going into functions when they are called. If a line contains multiple statements, execution moves to the next statement on the line.

Restrictions

The **step** command does not stop inside object code functions that do not have debugging information (functions compiled without the **-g** switch).

The **step** command does not stop in functions that initialize statics.

See also

next, stepout, stepi

“Using Workspace break levels” on page 160

“Running, continuing, and stepping” on page 163

stepi

steps execution in machine instructions by statement, entering functions

Command syntax

stepi
stepi *number*

Description

<< *none* >>	Executes a single machine instruction and then stops execution.
number	Executes the specified number of machine instructions and then stops execution.

Usage

Use the **stepi** command to single-step through the machine instructions in your program, going into functions when they are called. If a line contains multiple statements, execution moves to the next statement on the line.

See also

listi, nexti, step, stopi

"Using Workspace break levels" on page 160

"Running, continuing, and stepping" on page 163

"Debugging machine instructions" on page 186

stepout

continues execution until the current function returns

Command syntax

stepout

Description

<< *none* >>	Continues execution until the current function returns and then stops execution at the next statement in the calling function.

Usage

Use the **stepout** command to move execution to the point where the current function returns. This command is particularly useful if you inadvertently step into a function and want to continue stepping through the calling function.

See also

next, step

“Using Workspace break levels” on page 160

“Running, continuing, and stepping” on page 163

stop

sets a breakpoint

Command syntax

stop
stop if *cond*
stop at *line*
stop at *line* **if** *cond*
stop in *func*
stop in *func* **if** *cond*

Description

<< *none* >>	Sets a breakpoint at the current location. Displays a stop sign next to the line containing the breakpoint in the Source area.
if *cond*	Creates a break level and stops execution if *cond* is true, where *cond* is a Boolean expression.
at *line*	Sets a breakpoint at the specified line in the current file.
at *line* **if** *cond*	Sets a breakpoint at the specified line in the current file if *cond* is true, where *cond* is a Boolean expression.
in *func*	Sets a breakpoint at the first line of the specified function.
in *func* **if** *cond*	Sets a breakpoint at the first line of the specified function if *cond* is true, where *cond* is a Boolean expression.

Usage

Use the **stop** command to set a breakpoint in your program's code. When the breakpoint is encountered, execution is interrupted and a break level is created. To continue execution after the breakpoint, use the **cont** command.

You can also set a conditional breakpoint with the **when** command. To remove a breakpoint, use the **delete** command. To view a list of all breakpoints, use the **status** command.

See also **cont, delete, status, stopi, when**

“Setting breakpoints and watchpoints” on page 150

“Setting actions” on page 155

“Examining and deleting debugging items” on page 158

“Using Workspace break levels” on page 160

stopi

sets a breakpoint at a machine instruction

Command syntax

stopi
stopi [at] *address*

Description

<< *none* >>	Sets a breakpoint on the current location's address.
[at] *address*	Sets a breakpoint on the specified address. Stops execution whenever the byte at the specified address is modified. The *address* argument must be specified as a numeric string.

See also

listi, nexti, stepi, stop

"Setting breakpoints and watchpoints" on page 150

"Setting actions" on page 155

"Examining and deleting debugging items" on page 158

"Debugging machine instructions" on page 186

unalias

removes an alias for a command

Command syntax

unalias *name*

Description

name	Deletes the the alias specified by *name*.

Usage

Use the **unalias** command to delete an alias that you no longer want to use.

Example

If you have an alias named **p** that invokes the **print** command, you can delete the alias with the following command:

```
pdm -> unalias p
```

See also

alias

"Using aliases for Workspace commands" on page 136

unsetenv

removes a variable from the program's environment

Command syntax

unsetenv *variable*

Description

variable	Removes the definition of *variable* from the system environment.

Usage

Use the **unsetenv** command to remove a variable from the program's system environment. The **unsetenv** command is analogous to the similarly named shell command.

The **unsetenv** command affects only your program's environment variables. It does not affect the environment variables used by CenterLine-C++ to control its own operations.

The environment is an array of strings that is made available to the program through the global **environ** variable and the **envp** parameter, which is passed as the third argument to the **main()** function. By convention, each string has the format **name=value**, where the **value** part is optional.

Warnings

If **unsetenv** is called from a break level, it will alter the value of the global **environ** variable, but not the **envp** parameter passed to **main()**. This problem also occurs with the **putenv()** function.

Changing the **EDITOR** or **DISPLAY** shell variables with **unsetenv** does not affect which editor or display screen CenterLine-C++ uses.

See also

printenv, setenv

"Customizing environment variables" on page 200

up

moves up the execution stack

Command syntax

up
up *number*

Description

<< *none* >>	Moves the current scope location up one level on the execution stack. The Source area shows file scoped to location and highlights it with an arrow.
number	Moves the current scope location the specified number of levels up the execution stack.

Usage

Use the **up** command to move the current scope location up the execution stack, toward the top level of the Workspace and away from the current break level.

The scope location is the point at which all variables, types, and macros are scoped. When a break level is generated, the scope location is set to the point at which execution was interrupted.

When at a break level, use the **where** command to display the execution stack. Use the **whereami** command to display the break location and the current scope location.

The **cont** command can be used to continue execution, and the **reset** command can be used to return to a previous break level or to the top level of the Workspace without continuing execution.

See also

cont, down, reset, where, whereami

"Using Workspace break levels" on page 160

"Moving in the execution stack" on page 166

use

displays or sets the directory search path

Command syntax

use
use *pathname ...*

Description

<< *none* >>	Displays the current directory search path.
pathname ...	Sets the list of directories to be searched to the specified pathname. If more than one pathname is listed, they must be separated by spaces. The directories can be specified as absolute or relative pathnames.

Usage

Use the **use** command to set the list of directories to be searched when a filename is given to the **debug, edit,** and **list** commands.

The **use** command does not provide a search path for loading **#include** files.

See also

cd, debug, edit, list

"Customizing your startup file" on page 191

whatis

lists all uses of a name

Command syntax

whatis *name*

Description

name	Displays all uses of the specified name as a function, variable, class / struct / union tag name, enumerator, type definition, or macro definition.

Usage

Use the **whatis** command to display all uses of an identifier name. An identifier name is a name for a function, variable, enumerator, class / struct / union tag name, type definition, or macro definition.

CenterLine-C++ first displays all uses of the name within scope at the current scope location, followed by all uses of the name not within scope. The order of the listing represents the order in which the specified name is resolved when it is used.

Example

In the following example, **Rectangle** is a struct:

```
pdm 8 -> whatis Rectangle
    struct Rectangle {
    struct __mptr *__vptr;
    struct Shape OShape;
    struct Point origin;
    int filled;
    short *row;
    short *col;
    struct Point extent;
};
```

See also

dump, display, help, list, man, print, whereis

“Using Workspace break levels” on page 160

“Examining data structures” on page 169

when

executes specified commands

Command syntax

when
when if *cond*
when [at] *line*
when [at] *line* **if** *cond*
when in *func*
when in *func* **if** *cond*

Description

<< *none* >>	Executes commands at current location.
if *cond*	Executes commands at current location if *cond* is true, where *cond* is a Boolean expression.
[at] *line*	Executes commands when the specified line in the current file is reached.
[at] *line* **if** *cond*	Executes commands when the specified line in the current file is reached if *cond* is true, where *cond* is a Boolean expression.
in *func*	Executes commands at the first line in the specified function.
in *func* **if** *cond*	Executes commands at the first line in the specified function if *cond* is true, where *cond* is a Boolean expression.

Usage

Use the **when** command to set debugging actions.

After you issue the **when** command, CenterLine-C++ prompts you for the commands to be executed. These commands can include calls to functions that are defined in the program.

By default, CenterLine-C++ remains stopped after executing the commands specified with **when**. If you want your program to continue after executing the commands, you must specify the **cont** command as the last one.

Example

Here is an example of how to use the **when** command:

```
pdm 4 -> when at 5 if i == 100
```

Then type commands to be executed (one per line). Typing "." or "end" completes the sequence.

```
when -> printf("in func : %d\n", i);
when -> i = 200;
when -> cont
when -> .
pdm 5 ->
```

See also

"Setting breakpoints and watchpoints" on page 150

"Setting actions" on page 155

"Examining and deleting debugging items" on page 158

"Using Workspace break levels" on page 160

where

displays the execution stack

Command syntax

where
where *number*

Description

<< *none* >>	Displays a traceback of the execution stack, starting from the location where the execution has stopped.
number	Displays a traceback of only the specified number of functions on the top of the execution stack. The most recent routines called are at the top of the stack.

Usage

Use the **where** command to display a traceback of the execution stack.

When execution is stopped, it is often useful to see a full stack trace with arguments. The **where** command displays the formal parameters of functions that contain debugging information.

See also

cont, down, up, whereami

"Using Workspace break levels" on page 160

"Moving in the execution stack" on page 166

whereami

displays the current break and scope locations

Command syntax

whereami

Description

<< *none* >>	Displays the current break and scope locations.

Usage

Use the **whereami** command to list the current break location and the current scope location. This is particularly useful for finding where you are once you have moved up or down the execution stack while at a break level.

Break location

The break location is the point at which execution stopped when the break level was entered.

Scope location

The scope location is the point to which variables, functions, and types are scoped. When a break level is entered, it is set to the break location. It can be changed to different locations on the execution stack with the **up** and **down** commands.

Display of locations

If you have not moved up or down in the execution stack while at a break level, the scope location and the break location are the same. The **whereami** command displays that location in the Source area, scrolling the display if necessary.

If you have moved up or down in the execution stack, the scope location is displayed in the Source area and the break location is shown in the Workspace.

NOTE If the **whereami** command appears not to respond as you expect, keep the following in mind:

- The break location is only displayed in the Workspace when the break location is different from the scope location.
- The Source area will only change if the current scope location is not already displayed there.

See also

cont, down, up, where

"Using Workspace break levels" on page 160

"Moving in the execution stack" on page 166

whereis

lists the locations where a name is declared or defined

Command syntax

whereis *name*

Description

name	Lists the locations where a name is declared or defined; lists only global and top-level static declarations.

Usage

Use the **whereis** command to list locations where a symbol is declared or defined as a global or top-level static.

Example

In the following example, the name **Point** is declared in three files.

```
pdm (break 1) 83 -> whereis Point
File shapes.C:
struct Point {
 int x;
 int y;
};

File rect.C:
struct Point {
 int x;
 int y;
};

File main1.C:
struct Point {
 int x;
 int y;
};
pdm (break 1) 84 ->
```

See also

list, display, whatis

“Using Workspace break levels” on page 160

“Examining data structures” on page 169

Appendix A

GNU General Public License

This appendix contains the GNU General Public License, which applies to the CenterLine GNU Debugger (pdm) and the CenterLine C preprocessor (clpp).

GNU General Public License

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1 This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2 You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3 You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4 You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5 You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6 You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7 Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8 If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9 If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10 The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11 If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12 BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13 IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND / OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Applying These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> ONE LINE TO GIVE THE PROGRAM'S NAME AND AN IDEA OF WHAT IT DOES
> Copyright (C) 19YY NAME OF AUTHOR
>
> This program is free software; you can redistribute it and / or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19yy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.
>
> SIGNATURE OF TY COON, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V