ObjectCenter User's Guide

Version 2.1.1

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

Using this book

What this manual is about

This manual describes how to use ObjectCenter™ to develop C++ and C applications. It describes how to

- Bring your program into ObjectCenter
- Tailor the ObjectCenter programming environment to meet your immediate programming objectives and match your own programming methodology
- Fix static errors by using load-time error checking
- Interactively debug source and object files using run-time error checking and source-level debugging facilities in component debugging mode
- Interactively debug fully linked executables using source-level debugging facilities in process debugging mode
- Comprehend your code using program and data visualization facilities
- Customize ObjectCenter to your particular requirements

For your convenience, the Index in this *User's Guide* contains entries for the *ObjectCenter Reference* as well as the *User's Guide*.

What you should know before starting

This manual does not assume any previous knowledge of ObjectCenter. However, the *ObjectCenter Tutorial* provides the best introduction to the product and gives you a feel for what it is like to work in the ObjectCenter programming environment. If you haven't already gone through the tutorial, we strongly recommend that you begin there.

This manual does assume that you are familiar with the C++ and C languages. It does not attempt to teach C++ or C programming. It also assumes that you are familiar with UNIX® and the environment in which you will be running ObjectCenter (such as the Motif® or OPEN LOOK® graphical environment under the X Window System™).

For more information

The *ObjectCenter Tutorial* is a hands-on introduction to ObjectCenter. The tutorial leads you step by step through programming scenarios. It is the best place to start learning about ObjectCenter.

The *ObjectCenter Reference* provides a complete reference to Version 2.1.1 of ObjectCenter. The *ObjectCenter Reference* is an alphabetical reference that contains entries for topics as well as for Workspace commands and predefined functions. Some examples of topics are as follows: ANSI C, built-in functions, debugging, language selection, options, environment variables, C library functions, templates, and X resources. The *ObjectCenter Reference* assumes that you are already familiar with the material presented in the *User's Guide*.

The C++ language supported by ObjectCenter is Release 3.0 of the AT&T C++ Language System. The ObjectCenter documentation also includes two manuals shipped by AT&T in support of Release 3.0 of C++:

- 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.

Relevant excerpts from additional AT&T documents are also included in the ObjectCenter documentation in the Manual Browser.

The *ObjectCenter Platform Guide* describes system requirements and information specific to a particular platform. The *Platform Guide* is available online as an appendix to the *ObjectCenter Reference*.

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

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

Documentation conventions

Unless explicitly noted, this manual uses the following typographical conventions:

literal names	Bold words or characters in commands descriptions represent words or values that you must use literally.
*user-supplied value*s	Italic words or characters in command descriptions represent values that you must supply. Italic words in text also indicate the first use of a new term, or 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 in commands indicate that you can repeat the preceding item one or more times.

NOTE You will find notes like these throughout this manual. These notes highlight information of special importance.

TIP: Using tips

Tips provide additional information on special issues, such as troubleshooting and workarounds.

Contents

user_guide.book : user_guideTOC.doc xiv Mon Jun 5 13:07:07 1995 user_guide.book : user_guideTOC.doc xv Mon Jun 5 13:07:07 1995

List of Tables

List of Figures

List of Tips

Chapter 1 An overview of the ObjectCenter programming environment

This chapter introduces you to ObjectCenter by answering the following questions:

- *What is ObjectCenter?*
- *How does ObjectCenter improve your productivity?*
- *How do you work in ObjectCenter?*

At the end of the chapter, we describe the organization of the rest of the book.

What is ObjectCenter?

ObjectCenter is designed to enhance your productivity as a programmer, improve the quality of the code you deliver, and generally make it easier for you to tackle the day-in and day-out tasks of prototyping, compiling, debugging, testing, polishing, and maintaining C++ and C code. In short, ObjectCenter is all about enhancing your satisfaction in producing on-time, quality software.

Increased productivity, code quality, and work satisfaction are the natural consequence of a programming environment that gives you quick, seamless access to a full set of integrated tools and capabilities for interactive C++ and C programming.

A comprehensive toolset

ObjectCenter offers a comprehensive set of tools for C++ and C programming:

- An incremental linker / loader
- A load-time code checker
- A run-time program checker
- A component debugger for source and object files
- A process debugger for externally linked executables
- An interpreter for the full C++ and C languages
- A C++ translator, which converts C++ source code to C source code prior to compilation

Graphical access to functionality

ObjectCenter provides a Graphical User Interface (GUI) that allows you to access ObjectCenter's toolset easily through the work areas shown in the following table.

The ObjectCenter GUI enables you to perform certain basic actions, such as listing or editing code, in those primary work areas in which performing those actions would be useful.

Primary Work Area	**Allows You To**
Main Window	Access debugging functionality and control your session. In the Source area, display source code. In the Workspace, invoke ObjectCenter commands and enter C++ and C statements.
Project Browser	Manage the files and libraries that make up your current project and, in the Contents window, examine definitions contained in each file.
Error Browser	Examine and manage load-time and run-time errors and warnings.
Cross-Reference Browser	Examine the static dependency structure in your program (references to functions and global variables).
Data Browser	Examine data graphically.
Inheritance Browser	Examine the inheritance relationships among the classes you have loaded.
Class Examiner	Examine the members of a class.
Options Browser	Tailor various aspects of the ObjectCenter environment.
Manual Browser	View ObjectCenter manuals online.
Thread Browser	Examine the state of threads and lightweight processes in your program.

Within primary work areas, the GUI enables you to perform an action in more than one way so you can perform it in the way you find most convenient. For example, the Project Browser enables you to instrument an uninstrumented file by selecting a choice on the pull-down menu, the pop-up menu, or a button on the Control Panel. This book discusses at least one way to perform each action on the primary work areas. By looking at the GUI, you can find all the ways to perform an action.

An integrated, open architecture

Underneath ObjectCenter's toolset and GUI is an architecture that allows it to be much more than just a collection of programming tools. ObjectCenter's architecture consists of a set of cooperating components that communicate through the CenterLine™ message server. Each process contributes a specific aspect of the integrated functionality that makes ObjectCenter a programming environment:

- The CenterLine Message Server (CLMS)

  CLMS is a multicast message delivery service for exchanging data among the other ObjectCenter processes.

- A GUI (graphical user interface)

  You can choose either the Motif or OPEN LOOK environment. The GUI components represent each of the primary work areas of the GUI listed in 'Graphical access to functionality' on page 3.

  The GUI provides ease of use for ObjectCenter's integrated toolset. Each component of the GUI provides a distinct work area for accessing ObjectCenter's functionality.

- The CenterLine Engine, which you can choose to be either *component debugging mode* (cdm) or *process debugging mode* (pdm)

  When in component debugging mode, the CenterLine Engine operates on source and object files. It incorporates a full C++ and C interpreter, a load-time code checker, and a run-time program checker.

  When in process debugging mode, the CenterLine Engine operates on externally linked executables.

- Edit Server

  The Edit Server translates edit requests and responses between ObjectCenter and your editor. The Edit Server shipped with ObjectCenter provides tight integration with **vi** and GNU **emacs**.

Figure 1 shows the relationships among the various components of ObjectCenter's integrated, open architecture. Two-headed arrows indicate the interfaces between the **clms** process and other ObjectCenter processes.

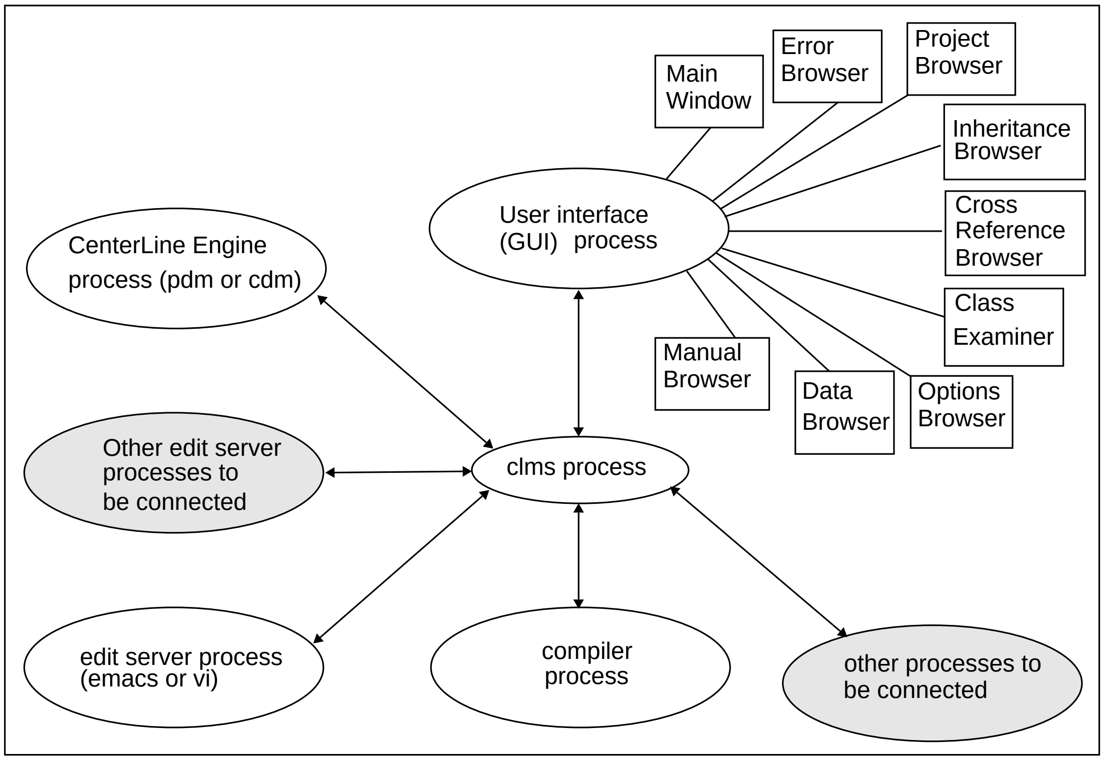

Main Window
Error Browser
Project Browser
Inheritance Browser
User interface (GUI) process
Cross Reference Browser
CenterLine Engine process (pdm or cdm)
Class Examiner
Manual Browser
Data Browser
Options Browser
Other edit server processes to be connected
clms process
edit server process (emacs or vi)
compiler process
other processes to be connected

Figure 1 ObjectCenter's Architecture: A Conceptual Illustration

Accessing ObjectCenter's open architecture

Because ObjectCenter's architecture consists of a set of cooperating components that communicate through the CLMS, ObjectCenter's architecture is open for integration with other programming tools using the CenterLine Application Program Interface (API).

The CenterLine API allows third-party software vendors to easily integrate their tools with ObjectCenter. In addition, you can integrate your own development tools, such as your editor or version control tool. For more information, see the **CenterLine API** and **CLIPC** entries in the *ObjectCenter Reference*.

How does ObjectCenter improve your productivity?

To see how ObjectCenter delivers on its promise of on-time, quality code, consider the productivity gains from each of ObjectCenter's programming features.

Full support for both C++ and C languages

ObjectCenter supports development in both C++ and C.

You can mix C files with C++ files in your program or develop either straight C or straight C++ programs. This dual language support is especially helpful if you are working on a project that is moving from C to C++.

ObjectCenter supports the full C++ language, as defined by the *AT&T C++ Language System Product Reference Manual* for Release 3.0 of the AT&T C++ Language System. Particularly important with ObjectCenter's support for Release 3.0 of AT&T C++ is support for class and function templates.

ObjectCenter also supports the full C language, as defined by Kernighan and Ritchie (K&R), and has further support for the ANSI C standard.

Incremental linking and loading

ObjectCenter cuts through the delay of your usual edit, compile, link, and debug cycle.

When you make a change to your source code, ObjectCenter's incremental linker / loader recompiles, relinks, and reloads only the files affected by your change. You get rapid turnaround because you no longer need to wait for a relink of your entire program.

Demand-driven code generation

Demand-driven code generation is the process of selectively generating code according to whether the code is actually used. ObjectCenter's version of the C++ translator supports demand-driven code generation outside the environment with the **-dd=on** and **-dd=off** switches to the **CC** command. Similarly, you can specify demand-driven code generation for both source and object files within the ObjectCenter environment with the **-dd=on** and **-dd=off** switches to the **load** command.

Within the environment, using demand-driven code generation can improve load-time significantly for source files and object files that require compilation.

See the **demand-driven code generation** entry in the *ObjectCenter Reference*.

Precompiled header files

ObjectCenter provides its own version of the AT&T C++ Language System, including the translator (cfront), which translates C++ code to C code, prior to compilation. To supplement the language system provided by AT&T, ObjectCenter provides a facility to keep track of header files that have been compiled and skip them whenever possible on subsequent compilation of the same program.

See the **precompiled header files entry** in the *ObjectCenter Reference*.

Templates

Templates are the mechanism in C++ for supporting parameterized types. Parameterized types allows you to implement generic code for a type and them implement that type with different parameters.

If you are already familiar with C++ templates, see the “Summary of Usage” section in the **templates** entry of the *ObjectCenter Reference* for a quick description of their implementation in ObjectCenter. In case you are not familiar with templates, the **templates** entry also contains background information about how templates are defined in the C++ language.

Load-time error checking

ObjectCenter streamlines the otherwise tedious tasks of locating and fixing static problems in your code.

When you load a source file, ObjectCenter’s load-time code checker acts as a super-lint and automatically checks for static errors. The load-time code checker identifies syntax errors that you need to correct to get to compilable code and also identifies potentially dangerous, although legal, code that you need to clean up to have a polished, maintainable program. Because you locate and identify static problems simply by loading a file, ObjectCenter makes tracking syntax errors and cleaning code largely automatic.

Run-time error checking

ObjectCenter automatically locates silent, run-time bugs that can hide even in apparently well-behaved programs.

When you execute a program loaded into ObjectCenter as source and object files, ObjectCenter's run-time program checker automatically checks for over 80 run-time warnings and errors. Simply by running your program in ObjectCenter, you are assured of catching silent bugs early on—bugs that otherwise might not be found until quality assurance testing late in the programming cycle or that might even go out with your finished program.

Interactive debugging modes

ObjectCenter allows you to debug interactively and makes it easy for you to find a known bug—whether in source code, object code, or in an externally linked executable. ObjectCenter offers two distinct debugging modes: cdm and pdm.

Component debugging mode

Component debugging mode (cdm) offers superior debugging capabilities over run-time checking and process debugging mode, described in the next section. In component debugging mode, you debug source and object files that you load as components of an ObjectCenter project. Component debugging mode gives you a full range of interactive debugging features: breakpoints, watchpoints, stepping through code, and tracing execution. You can extend breakpoints and watchpoints by specifying customized actions. In component debugging mode, you need not be working on full executables.

Process debugging mode

In process debugging mode (pdm), you debug externally linked executables. Process debugging mode offers breakpoints, watchpoints, and stepping. In addition, process debugging mode also allows you to step execution in machine instructions.

Graphic visualization of file, function, and data structures

ObjectCenter improves your understanding of your code (code comprehension) through graphical views of program elements.

The Contents window of the Project Browser shows the definitions contained in your files, the Cross-Reference Browser shows the calling hierarchy in your program, and the Data Browser shows the data structures in your code. With the Inheritance Browser, you can examine the inheritance relationships among the classes you have loaded, and, with the Class Examiner, you can examine class members.

Full C++ and C interpreter

ObjectCenter enables you to execute independent modules for interactive unit testing and also to execute isolated C++ statements for prototyping as you go.

When you are in component debugging mode, ObjectCenter's interpreter allows you to immediately execute any part of your application or brand new code that you enter in the Workspace. The interpreter combines the functionality of an interactive unit tester and an interactive prototyper.

ObjectCenter supports both Kernighan and Ritchie (K&R) C and the ANSI standard C language. The default setting depends on the underlying compiler in use, which is different for each platform. The default setting is likely to be one that you are accustomed to on your platform.

Full flexibility to tailor performance

ObjectCenter allows you to maximize the performance characteristics that matter the most to you at a particular time: speed, space, automatic error checking, or debugging facilities.

ObjectCenter gives you many choices in how to set up your work at any point. You can either target an externally linked executable or else load source and object files as components of a project. If you target an executable, you have the choice of the executable alone, the executable with a corefile, or the executable as a running process.

If you load source or object components as a project, you have the choice of the composition of the project—anywhere from all object code to all source code. If you load object code, you can choose to load object code with or without debugging information. And once object code is loaded, you can either instrument it to enable run-time error checking or leave it as regular object code.

Using these various choices, at any point, you can set up your work in ObjectCenter to fit with your current programming objectives. You can tailor ObjectCenter to the balance that suits you among the following performance factors: the amount of static and run-time error checking, range of source-level debugging features, speed of startup, speed of execution, or amount of memory required.

Compatibility with standard UNIX tools

ObjectCenter provides programming functionality consistent and compatible with your standard UNIX programming tools, such as **make**, **lint**, **CC**, **cc**, **dbx**, **vi**, and **emacs**.

Since ObjectCenter is designed to work with standard UNIX tools, it is fully compatible with the compilers, preprocessors, and makefiles that you currently use. By making the entire programming process thoroughly interactive and at the same time completely compatible with your existing UNIX tools, ObjectCenter enhances your own programming style without imposing a programming methodology.

Graphical user interface

To access the power of the toolset, ObjectCenter offers a selectable, easy-to-learn, easy-to-use GUI.

When you start ObjectCenter, you can select one of three user interfaces: A Motif GUI, an OPEN LOOK GUI, or a nongraphical user interface (Ascii ObjectCenter). Both the Motif and OPEN LOOK GUIs provide a standard windowing interface to ObjectCenter for users of the X11™ window systems. You choose which standard you are most comfortable with.

Extensive online help

ObjectCenter offers a full range of online help: extensive context-sensitive spot help for every element of the GUI, a **Help** menu with topical help, summary help on commands in the Workspace, and a Manual Browser with a set of ObjectCenter documentation.

Ascii ObjectCenter

Most features available when running ObjectCenter with the Motif or OPEN LOOK GUI are also available in a nongraphical form with Ascii ObjectCenter. Typically, you use Ascii ObjectCenter if you are using it from a nongraphical terminal or over a phone line. This manual concentrates on describing the graphical versions of ObjectCenter. Differences between the graphical version of ObjectCenter and Ascii ObjectCenter are described in Chapter 9, “Using Ascii ObjectCenter.”

Customizable environment

ObjectCenter allows you to customize and extend the programming environment to suit your own work style, preferences, and additional programming tools.

You customize the ObjectCenter environment by using initialization files that configure startup behavior, X resource settings for configuring the GUI, and options for settings that control ObjectCenter’s behavior during a session.

To integrate new tools into ObjectCenter’s open architecture, you use the CenterLine Application Program Interface (CenterLine API) mechanism as described in the **CenterLine API** entry in the *ObjectCenter Reference.*

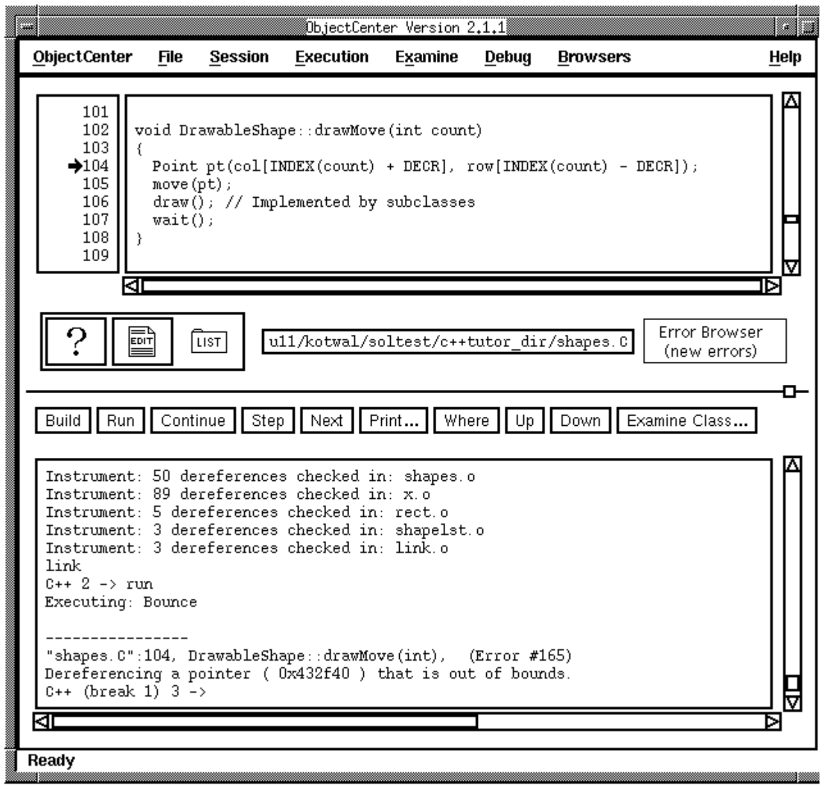
How do you work in ObjectCenter?

Now that you have surveyed the wide range of functionality that ObjectCenter offers you, let's take a quick walk through a typical working session in ObjectCenter.

Beginning your session

At startup, you choose either a Motif or an OPEN LOOK GUI and decide on the debugging mode (process or component). Let's assume you take the default settings, which will put you in component debugging mode.

The first thing you see when you start ObjectCenter is the Main Window, which is the central work area in ObjectCenter. It consists of the Source area, where source code is displayed, and the Workspace, where you can enter commands and C++ or C statements. Here is an illustration of the Main Window:



```
void DrawableShape::drawMove(int count)
{
  Point pt(col[INDEX(count) + DECR], row[INDEX(count) - DECR]);
  move(pt);
  draw(); // Implemented by subclasses
  wait();
}
```

```
u11/kotwal/soltest/c++tutor_dir/shapes.C
```

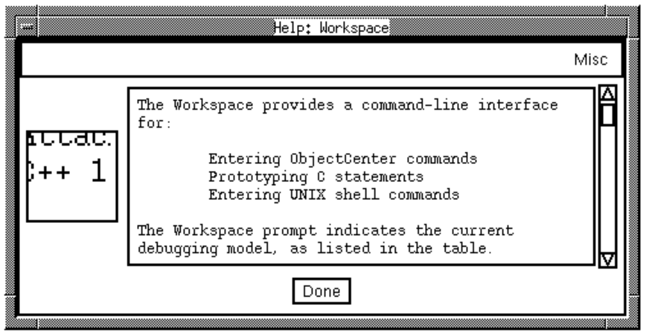
Error Browser
(new errors)

```
Instrument: 50 dereferences checked in: shapes.o
Instrument: 89 dereferences checked in: x.o
Instrument: 5 dereferences checked in: rect.o
Instrument: 3 dereferences checked in: shapelst.o
Instrument: 3 dereferences checked in: link.o
link
C++ 2 -> run
Executing: Bounce

----------------
"shapes.C":104, DrawableShape::drawMove(int),  (Error #165)
Dereferencing a pointer ( 0x432f40 ) that is out of bounds.
C++ (break 1) 3 ->
```

Since all components of ObjectCenter use the Source area to display source code, you usually keep the Main Window visible throughout your ObjectCenter session.

Using spot help to learn about ObjectCenter

As you work in ObjectCenter, you can rely on ObjectCenter's extensive spot help to get answers to your immediate questions. For example, to find out more about the Workspace, you move the cursor into the Workspace region and press the **F1** or **Help** key. A help window appears describing what the Workspace is and how to use it. Here is an example of a help window:



```
The Workspace provides a command-line interface
for:

        Entering ObjectCenter commands
        Prototyping C statements
        Entering UNIX shell commands

The Workspace prompt indicates the current
debugging model, as listed in the table.
```

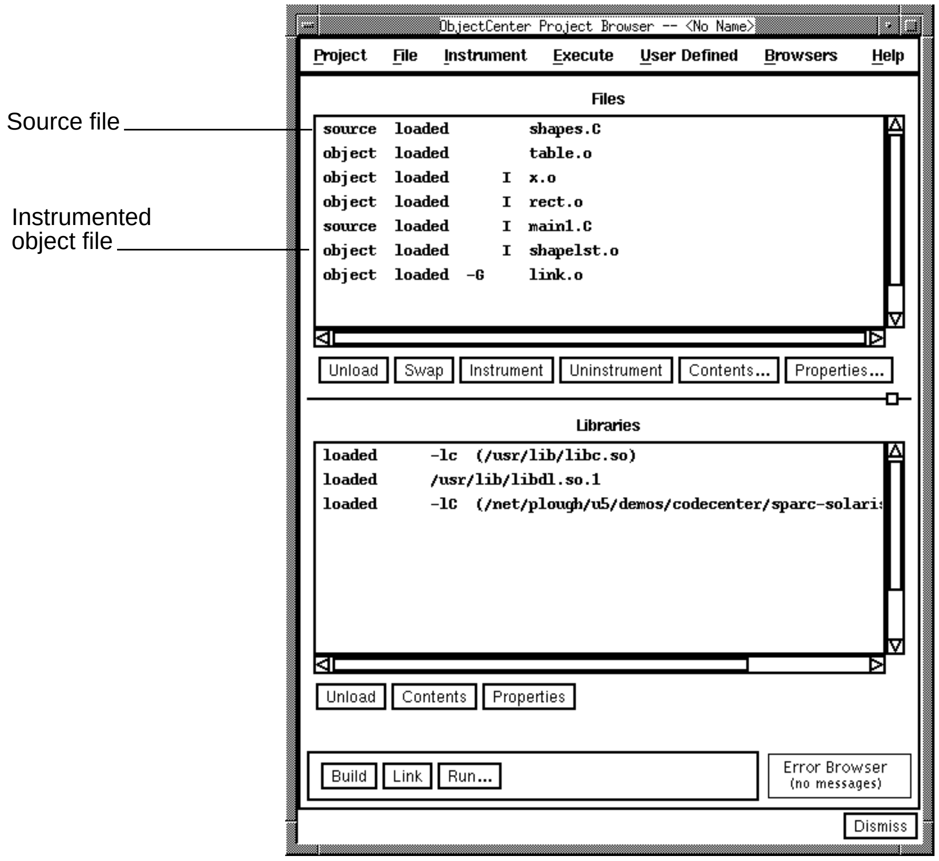
Getting your code into ObjectCenter and setting up your work

You begin working in ObjectCenter by getting your code into the programming environment and setting up your work in a way that best suits your present objectives. The files that you load for your session make up your current ObjectCenter project. Typically, you would begin by loading an existing project file or using ObjectCenter's **make** command to load a set of files.

At any time, you can tailor your project to have any combination of source and object files. For the best balance of speed, memory, and debugging capabilities, you ordinarily load only one or two files that you want to focus on as source and the rest as object files. Typically, you use ObjectCenter's **instrument** command to enable run-time error checking on the object files that you load. You use the Project Browser to examine and manage the files you have loaded.

NOTE The **instrument** command is not available on some platforms. Refer to the *ObjectCenter Platform Guide* to see if you can use it on your particular platform.

Here is an illustration of the Project Browser:



```
source  loaded        shapes.C
object  loaded        table.o
object  loaded      I  x.o
object  loaded      I  rect.o
source  loaded      I  main1.C
object  loaded      I  shapelst.o
object  loaded   -G    link.o
```

```
loaded      -lc  (/usr/lib/libc.so)
loaded      /usr/lib/libdl.so.1
loaded      -lC  (/net/plough/u5/demos/codecenter/sparc-solari
```

If you were debugging an externally linked executable, rather than loading source and object files, you would have placed ObjectCenter in process debugging mode and specified an externally linked executable as your debugging target.

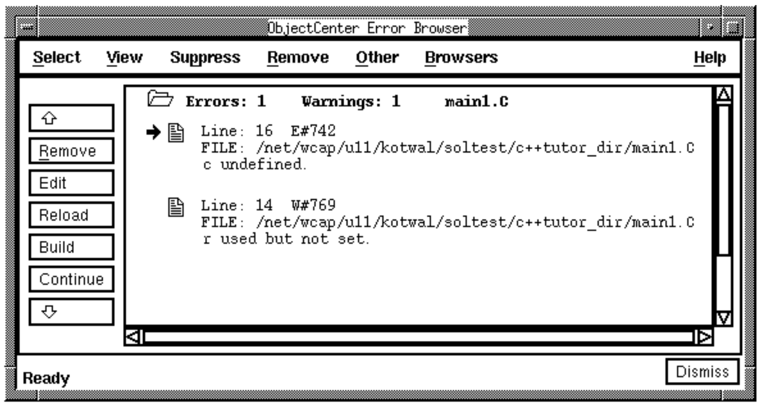
Working with templates

If your C++ code uses class or function templates (a feature introduced with Release 3.0 of AT&T C++), ObjectCenter loads and handles them transparently by default. You simply load this code into ObjectCenter the same as you would any other C++ code. (You do *not* explicitly load template declaration, definition, or map files.)

You only have to make explicit adjustments for working with templates if you want to customize how ObjectCenter handles them. For more information on working with templates, see the **templates** entry in the *ObjectCenter Reference*.

Fixing static errors

Because ObjectCenter performs extensive error checking on any source files that you load, the process of loading files often uncovers static errors. To deal with these load-time warnings and errors, you open the Error Browser and examine the static problems that ObjectCenter has identified in your code. Here is an example:



```
Errors: 1    Warnings: 1     main1.C
Line: 16  E#742
FILE: /net/wcap/u11/kotwal/soltest/c++tutor_dir/main1.C
c undefined.

Line: 14  W#769
FILE: /net/wcap/u11/kotwal/soltest/c++tutor_dir/main1.C
r used but not set.
```

ObjectCenter's integrated editor support allows you to edit static warnings and errors quickly. When you select a warning or error message icon, ObjectCenter invokes your editor, loads the file into the editor, and opens a window displaying the line with the warning or error ready for you to edit. After making all the corrections, you simply build your project, and ObjectCenter reloads and relinks only the files that have changed.

For less serious static problems, you can choose to suppress a warning and make the correction at a later cleanup phase.

If you load an object file that is out-of-date, ObjectCenter sends the file to **CC** and catches any resulting compiler error messages, which are also displayed and managed in the Error Browser.

Interactive debugging

After successfully getting your code into ObjectCenter, you either move directly into interactive debugging or use one of the Browsers to better understand your program. Let's assume that you decide to start debugging.

To take advantage of ObjectCenter's extensive run-time error checking, you use ObjectCenter's **run** command to invoke your program. ObjectCenter executes your program and automatically checks for run-time errors in source files and instrumented object files. If ObjectCenter encounters a run-time error, the **Error Browser** button indicates that you have a run-time error.

The run-time error is recorded in the Error Browser, and you can deal with the error there. To fix the bug, you take advantage of ObjectCenter's rapid turnaround with editor integration and incremental loading / linking facilities.

To extend your interactive debugging, you can use any of ObjectCenter's standard debugging capabilities, such as setting breakpoints, watchpoints, actions, and stepping through code. In addition to running the entire program, you can use ObjectCenter's C++ or C interpreter to directly call any part of the program. In this way you can test a function interactively with various arguments.

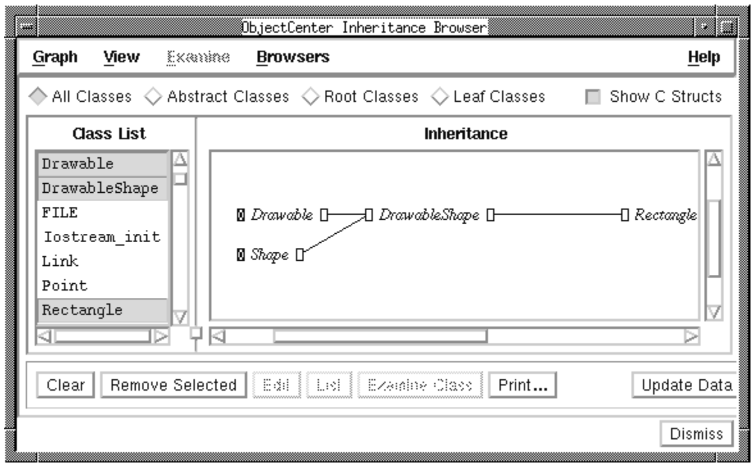
Visualizing your code

At any time, you can move between interactive debugging and exploring some aspects of your program's structure to better understand your code. You use ObjectCenter tools for code comprehension at the class, file, function, and data structure levels.

Visualizing at the class level

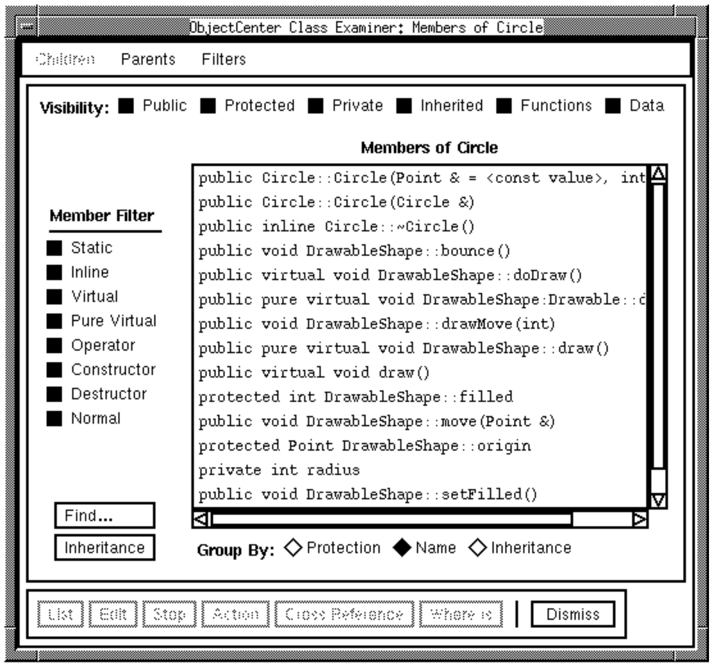
To better understand classes, you use the Inheritance Browser and the Class Examiner. The Inheritance Browser displays a graphic representation of the inheritance relationships among classes. The Class Examiner shows you the members of a class. In the Class Examiner, you can sort members by name, inheritance, or protection level. You can also filter class members by protection level and by a variety of data-member and function-member characteristics.

This is what the Inheritance Browser looks like:



```
Drawable
DrawableShape
FILE
Iostream_init
Link
Point
Rectangle
```

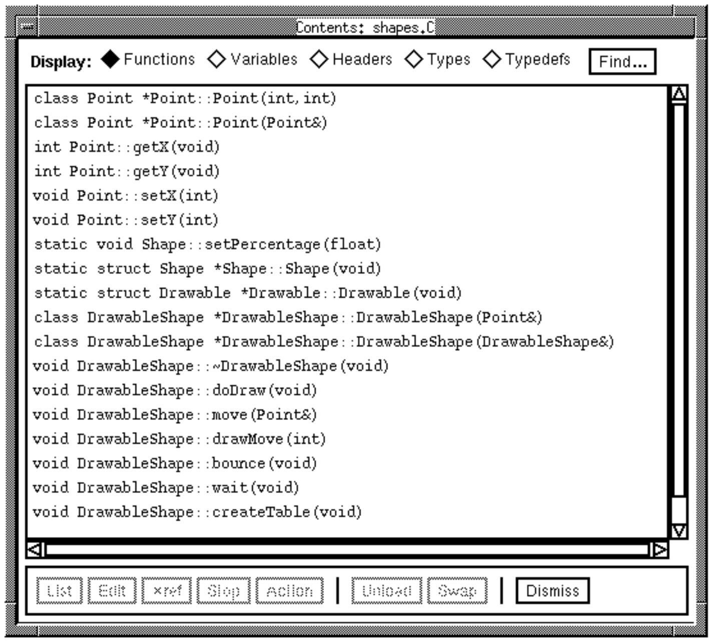
This is what the Class Examiner looks like:



```
public Circle::Circle(Point & = <const value>, int
public Circle::Circle(Circle &)
public inline Circle::~Circle()
public void DrawableShape::bounce()
public virtual void DrawableShape::doDraw()
public pure virtual void DrawableShape:Drawable::d
public void DrawableShape::drawMove(int)
public pure virtual void DrawableShape::draw()
public virtual void draw()
protected int DrawableShape::filled
public void DrawableShape::move(Point &)
protected Point DrawableShape::origin
private int radius
public void DrawableShape::setFilled()
```

Visualizing at the file level

To better understand the files and libraries that make up your ObjectCenter project (all the files and libraries you currently have loaded), you use the Contents window of the Project Browser. The Project Browser provides a way for you to examine the definitions of the functions, variables, headers, types, and typedefs contained in source files, object files, and libraries, as shown in the following illustration:

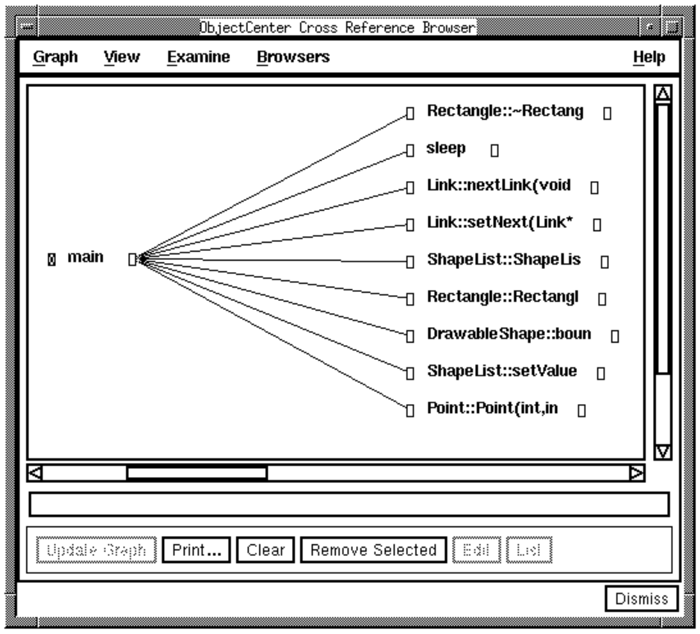


```
class Point *Point::Point(int,int)
class Point *Point::Point(Point&)
int Point::getX(void)
int Point::getY(void)
void Point::setX(int)
void Point::setY(int)
static void Shape::setPercentage(float)
static struct Shape *Shape::Shape(void)
static struct Drawable *Drawable::Drawable(void)
class DrawableShape *DrawableShape::DrawableShape(Point&)
class DrawableShape *DrawableShape::DrawableShape(DrawableShape&)
void DrawableShape::~DrawableShape(void)
void DrawableShape::doDraw(void)
void DrawableShape::move(Point&)
void DrawableShape::drawMove(int)
void DrawableShape::bounce(void)
void DrawableShape::wait(void)
void DrawableShape::createTable(void)
```

Visualizing at the function level

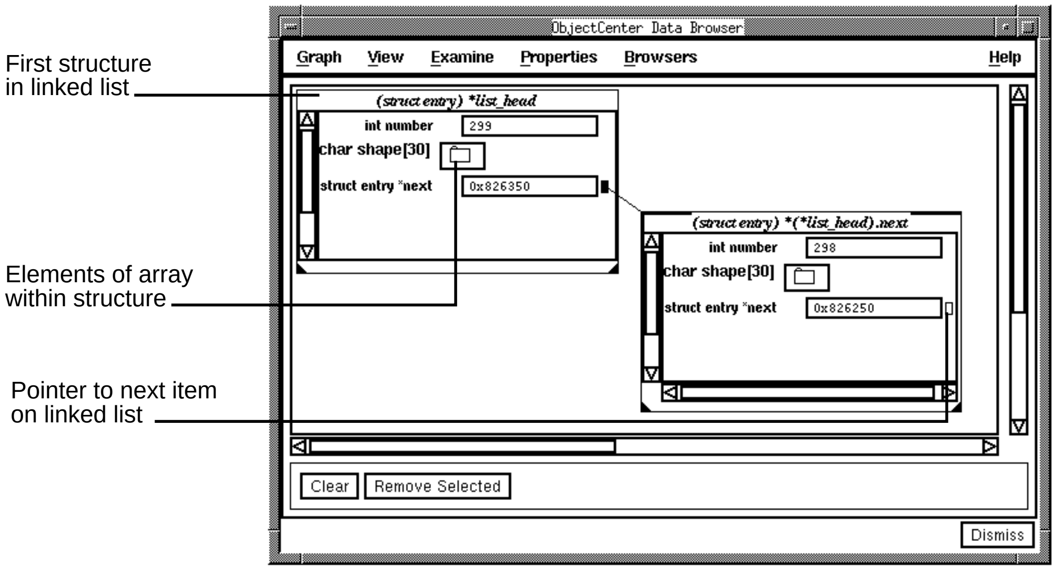
To better understand the calling structure for functions and references to global variables in your program, you use the Cross-Reference Browser. The Cross-Reference Browser graphically presents references to and from functions and references to global variables.

With the Cross-Reference Browser, you can also show the calling hierarchy of a function at any depth and filter the display to suit your needs at the moment, as shown in this example:



Visualizing at the data level

To better understand data structures, you use the Data Browser. The Data Browser displays a graphic representation of any data structure. Within complex data structures, you can follow pointers and examine substructures. For example, you can display a linked list and, at your discretion, display elements of an array within a struct that is a node in the linked list, as shown:


First structure in linked list
Elements of array within structure
Pointer to next item on linked list
ObjectCenter Data Browser
Graph
View
Examine
Properties
Browsers
Help
(struct entry) *list_head
int number
299
char shape[30]
struct entry *next
0x826350
(struct entry) *(*list_head).next
int number
298
char shape[30]
struct entry *next
0x826250
Clear
Remove Selected
Dismiss

More interactive development

With the context gained from code visualization, you continue to interactively modify and develop your program by using run-time error detection in conjunction with ObjectCenter's debugging facilities.

Once you are satisfied with the code in the files you have been focusing on and are ready for a final cleanup of the code, you can go back and unsuppress any load-time warnings that you suppressed earlier on. You can reload those files in source form and use the Error Browser to go quickly through your cleanup phase.

An all-day programming environment

As this typical session shows, within ObjectCenter you move fluidly from one phase of your programming cycle to the next: prototyping, developing, and bringing in new modules; tracking down a bug; studying an existing body of code; doing unit testing; or polishing up code that is functionally sound. ObjectCenter’s incremental linker/loader recompiles, relinks, and reloads only the files affected by a change. You get rapid turnaround because you no longer need to wait for a relink of your entire program.

Because you use ObjectCenter for all phases of your programming cycle, you typically start up ObjectCenter and leave it running throughout your workday. Rather than merely having a collection of individual programming tools that you start up for the need of the moment and then quit from, ObjectCenter gives you a programming environment that you want to keep resident day in and day out.

What’s Next?

The rest of this manual describes how to take full advantage of the features introduced in this overview. Each chapter covers a different aspect of working with ObjectCenter.

Chapter 2: ObjectCenter basics

This chapter covers fundamental operations for using ObjectCenter effectively: options at startup, accessing functionality, getting help, listing and editing source code, switching debugging modes, and quitting.

Chapter 3: Managing your code in ObjectCenter

This chapter presents various approaches to getting code into ObjectCenter, either as an externally linked executable or as source and object file components for a project (loading individual files, saving and loading a project file or image file, or using specially designed makefile targets). It shows how to tailor the way you set up your work to maximize the performance factors in a way that works best for you. It also explains Project Browser operations for managing project components (building, linking, swapping, setting project-wide and file-specific properties) and gives tips for working with large projects.

Chapter 4: Fixing static errors

This chapter covers ObjectCenter's load-time error checking. It describes how ObjectCenter checks for static errors and how to use the Error Browser to deal with load-time errors and warnings.

Chapter 5: Component debugging

This chapter covers the various aspects of interactively debugging source and object components. It explains what it means to run a program in component debugging mode, and it explains using the interpreter for unit testing and interactive prototyping. It covers run-time error checking and how to use it to track down bugs. The chapter explains the various debugging operations available (breakpoints, actions, stepping, and tracing) and covers working with them at break levels. It goes on to cover some special issues for component debugging: working with code generated by a preprocessor and debugging multiple processes.

Chapter 6: Process debugging

This chapter covers the elements of interactive debugging that are unique to debugging an externally linked executable and differentiates process debugging from the component debugging covered in the previous chapter. It discusses when process debugging mode is most effective and explains how to specify an executable as a debugging target. It describes specific operations that are different from those used for component debugging.

Chapter 7: Visualizing your code

This chapter covers four levels of code visualization: understanding your program at the file, function, class, and data structure levels. It describes how to use the Project Browser's Contents window, the Cross-Reference Browser, the Data Browser, the Inheritance Browser, the Class Examiner, the Source area, and a group of Workspace commands to help you understand your code better.

Chapter 8: Customizing ObjectCenter

This chapter describes how you can configure the ObjectCenter environment using initialization files that configure startup behavior, X resources settings for configuring the GUI, and options for settings that control ObjectCenter's behavior during a session.

Chapter 9: Using Ascii ObjectCenter

This chapter describes how to use Ascii ObjectCenter when you do not have access to a graphical workstation or when you do not want to incur the overhead of a GUI.

Chapter 2 ObjectCenter basics

This chapter describes the fundamental operations for using ObjectCenter effectively. It covers the following topics:

- *Starting ObjectCenter*
- *Getting help*
- *Using ObjectCenter's GUI*
- *Entering ObjectCenter commands in the Workspace*
- *Listing source code*
- *Editing source code*
- *Quitting ObjectCenter*

Starting ObjectCenter

Using the ObjectCenter startup command

You start up ObjectCenter by invoking the **objectcenter** startup command at the shell. This command takes two types of arguments: startup switches and configuration switches.

The ObjectCenter startup 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.

Specifying the GUI and debugging mode at startup

When you start ObjectCenter, you can use the switches listed in Table 1 to specify both the user interface and debugging mode for ObjectCenter.

Table 1 ObjectCenter Startup Switches

Startup Switch	Result
[-ascii | -motif | -openlook]	Specifies the user interface: nongraphical (Ascii ObjectCenter), the Motif GUI, or the OPEN LOOK GUI. On Sun™ systems, if no switch is used, ObjectCenter defaults to the OPEN LOOK GUI. On all other platforms, the Motif GUI is the default.
[**-cdm | -pdm**]	Specifies debugging mode: cdm (component debugging mode)—for working with a project comprising source, object, and library file components. pdm (process debugging mode)—for working with an externally linked executable alone or with a corefile or running process to which ObjectCenter attaches. If no switch is used, ObjectCenter starts in component debugging mode.

TIP: Setting the DISPLAY environment variable

Before running the Motif or OPEN LOOK version of ObjectCenter, make sure to follow the usual X conventions for setting up your **DISPLAY** environment variable. Otherwise, the GUI may not display the way you expect it to.

For a complete list of switches you can use from the command line to affect the GUI, see the **objectcenter** entry in the *ObjectCenter Reference*.

Other command-line switches

In addition to switches for specifying the user interface and the debugging mode, you can specify a variety of other attributes by using command-line switches when starting ObjectCenter. For example, you can specify:

- A file in which ObjectCenter will save all the input you typed in the Workspace during a session
- The size of your run-time stack
- Alternative system or local startup configuration files

For information about ObjectCenter command-line switches, see the **objectcenter** entry in the *ObjectCenter Reference*.

Libraries automatically loaded at startup

When starting, ObjectCenter automatically loads the standard C++ library, **libC.a**, and the standard C library, **libc.a**, and any libraries that these depend on. ObjectCenter(The exact list of these other libraries varies depending on your platform.) On architectures that support shared libraries, ObjectCenter might load the shared version of this library. For platform-specific information about using libraries in ObjectCenter, see the *ObjectCenter Platform Guide* for your platform.

Customizing your session at startup

You can tailor how ObjectCenter begins each session by editing the global startup file **ocenterinit** or your local startup files **.ocenterinit** and **.pdminit**. At startup, ObjectCenter reads these two files and sets up your session according to the commands contained in them. Since the local startup file is read last, commands in this file override conflicting commands in the global file. For more information about customizing these startup files, see 'Using ObjectCenter startup files' on page 213.

Getting help

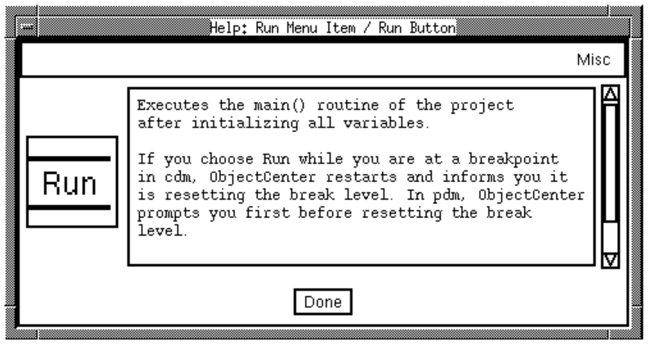
TIP: Using tips like this one

Both in this manual and in the *ObjectCenter Reference*, you can get help for troubleshooting and workarounds from tips boxes like this one. The title for each tip is listed in the List of Tips section following the Contents at the front of each manual. You can also use the **Troubleshooting** entry in the Index to look up tips based on symptoms.

Using spot help

The GUI versions of ObjectCenter provide an extensive system of spot 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 over the **Run** button below the Source area and press **F1**, you see the following help topic:



```
Executes the main() routine of the project
after initializing all variables.

If you choose Run while you are at a breakpoint
in cdm, ObjectCenter restarts and informs you it
is resetting the break level. In pdm, ObjectCenter
prompts you first before resetting the break
level.
```

Using the Help menu

In addition to spot help, the ObjectCenter GUI also offers help on a range of topics. You access this help through the **Help** menu in any primary window. For example, the **On Windows** help topic gives an overview of each of ObjectCenter’s primary windows.

Using Workspace help

The **help** command displays quick usage information about ObjectCenter commands. If you issue **help** without any arguments, ObjectCenter displays a summary of commands. If you issue **help** with the name of a command, ObjectCenter displays information about the specified command. For example:

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

Using the Manual Browser

The Manual Browser presents ObjectCenter documentation. You can open the Manual Browser from any primary window by displaying the **Browsers** menu and selecting **Manual Browser**. You can also open the online *Reference* by typing this in the Workspace:

```
-> man topic_name
```

Contacting CenterLine technical support

To get direct help with technical questions, offer suggestions, and report problems to CenterLine Software, you can send email to CenterLine Technical Support by using ObjectCenter's email facility. If possible, bug reports should include small examples of code that produce the bug.

To send email to CenterLine Software from within ObjectCenter, in the Main Window display the **ObjectCenter** menu and select **Send email**. A dialog box opens with the default setting for the Internet address of CenterLine Technical Support.

TIP: Correcting the support email address

If the ObjectCenter email messages are not being delivered properly, your system administrator should correct the installed support email address using the **cladmin** utility. For a short-term solution, you can specify a different mailing address by changing ObjectCenter's **email_address** option. See 'Using ObjectCenter options' on page 216 for more information.

Using ObjectCenter's GUI

In the GUI, you access ObjectCenter's functionality by using actions on menus, buttons, and value settings. In general, you operate in ObjectCenter by following the standard usage conventions that you would expect for a Motif or OPEN LOOK application. However, the following deserve special attention:

- Basic mouse actions
- Copying and pasting text
- GUI accelerators

NOTE This manual uses the word **select** to mean any of the following mouse-related actions:

- Causing a graphical object to get the focus
- Causing text to be highlighted
- Activating a menu item
- Activating a command button

Basic mouse actions

Table 2 describes the basic mouse actions you use in the ObjectCenter GUI. In instances where ObjectCenter's GUI uses mouse actions other than the standard conventions in Motif or OPEN LOOK, this book points these out explicitly. For example, mouse actions for pop-up menus and windows in the Source area are given in Table 3 on page 31.

Table 2 Basic Mouse Actions

To Do This	Use This Mouse Action
Select text	Press the Left mouse button and drag across the text.
Select a menu item, command button, or a graphical object	Click the Left mouse button while the pointer is on the item you want to select.
Display a pop-up menu or select a menu item from it	Click or press-and-drag with the Right mouse button.
Display a pull-down menu or select a menu item from it	For OPEN LOOK, click or press-and-drag with the Right mouse button. For Motif, click or press-and-drag with the Left mouse button.
Select the default action for a pull-down menu	Available for OPEN LOOK only. Click the Left mouse button on the menu button.
Change the default action for a pull-down menu	Available for OPEN LOOK only. Press the Control key and the Right mouse button. Select a menu choice to be the new default.

Using mnemonics for menu-bar selections

Each ObjectCenter primary window provides Motif-style mnemonics for almost every menu item on its menu bar. A menu item with a mnemonic has one of its letters underlined, usually the first one. To select an item with mnemonics:

1. Press the Meta key and the underlined-letter key at the same time, to display the menu.
2. Press the underlined-letter key of the menu item.

NOTE Mnemonics are not available for items that you can create and destroy on the fly during a session, such as items on the User Defined submenu of the ObjectCenter menu in the Main Window.

Main Window shortcuts with pop-up menus and windows

The Main Window and the Browsers provide pop-up menus and pop-up windows that give mouse-based shortcuts for accessing functionality. Because of their central place in the GUI, the Main Window's Source area and Workspace are particularly rich in these pop-up shortcuts.

The following table gives the shortcuts to use for accessing Source area and Workspace functionality through pop-up menus and windows:

Table 3 Main Window Shortcuts

Shortcut	Description
With the mouse pointer in the Source area, press the Right mouse button.	Displays the **File Options** menu. Gives operations to perform on the currently listed file. Also lets you set and delete debugging items.
With the mouse pointer in the Workspace, press the Right mouse button.	Displays the **Workspace Options** menu. Gives operations for using the Workspace. Also lets you delete debugging items.
With the mouse pointer over a line number at the left of the Source area, press the Right mouse button.	Displays the **Line Number Options** menu. Gives operations to perform on the specified line of code.
With an expression or identifier selected and the mouse pointer in the Source area or Workspace, hold the Shift key and press the Right mouse button. If no identifier is selected, with the mouse pointer on an identifier, hold the Shift key and press the Right mouse button.	Displays the **Expression Options** menu. Gives operations to perform on the selected expression or identifier.
With an expression selected and the mouse pointer in the Source area or Workspace, hold the Shift key and press the Middle mouse button. If no expression is selected, with the mouse pointer on an expression, hold the Shift key and press the Middle mouse button.	Displays the Whatis pop-up window. Shows the definition of the selected identifier.
With an expression selected and the mouse pointer in the Source area or Workspace, hold the Shift key and press the Left mouse button. If no expression is selected, with the mouse pointer on an expression, hold the Shift key and press the Left mouse button.	Displays the Print pop-up window. Shows the value that the selected expression evaluates to.

Copying and pasting selected text

Copying and pasting in ObjectCenter works as you would expect for any Motif or OPEN LOOK application.

Motif GUI

For the Motif GUI, you use the Middle mouse button to paste selected text (PRIMARY selection). To paste the CLIPBOARD selection, use the Paste key or Paste Menu (from the Workspace pop-up menu). If you try to paste a nonexistent CLIPBOARD selection, ObjectCenter pastes the PRIMARY selection.

OPEN LOOK GUI

For the OPEN LOOK GUI, you paste the CLIPBOARD selection using the Paste key or Paste Menu (from the Workspace pop-up menu). OPEN LOOK does not support pasting PRIMARY selections using the Middle mouse button.

By default, X applications use the PRIMARY selection rather than the CLIPBOARD selection. This means that you cannot copy and paste between an X application and the OPEN LOOK GUI. You can change this default behavior of copying and pasting by editing your **.Xdefaults** file. To customize X applications to use either type of selection, 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
```

NOTE This example binds the **L6** key to the copy function and the **L8** key to the paste function; however, you can substitute any keyboard symbols for these.

You can also customize the OPEN LOOK GUI so it can also paste PRIMARY selections. This allows you to cut and paste between the OPEN LOOK GUI and X applications. Add the following 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
```

NOTE This example binds the **Shift** and **L8** keys to the paste function; however, you can substitute any keyboard symbols for these.

Then load these definitions into your X resource database by using the UNIX **xrdb** command or by restarting your X server.

With these customizations, you can use either the standard copy and paste procedures of Motif (select text and click the Middle mouse button) or those of OPEN LOOK (select text and use the keys that you have bound to the copy and paste functions).

General GUI accelerators

The following features act as accelerators for actions that apply throughout the GUI.

Using selected text as an argument for an action

Many actions on the pull-down and pop-up menus use the X11 selection as a default argument without asking for confirmation. Before you choose an action on a menu, be aware of any text you may have selected. If the action can take an argument, ObjectCenter uses the selected text as the argument. See the next section "Dialog boxes" if you want to be able to confirm menu actions and arguments before you select them.

TIP: Dealing with surprising text in a dialog box input line

ObjectCenter's dialog boxes use the current X11 selection, no matter what the original source. If surprising text appears in a dialog box input line, check the current selection in other applications or X terminal windows.

Dialog boxes

By default, ObjectCenter does not display dialog boxes to confirm selection-based commands. If you want to be able to confirm menu actions and arguments before you select them, you can enable selection in pop-up dialog boxes in the GUI by using the following X resource setting:

```
ObjectCenter*ConfirmSelnUse: True
```

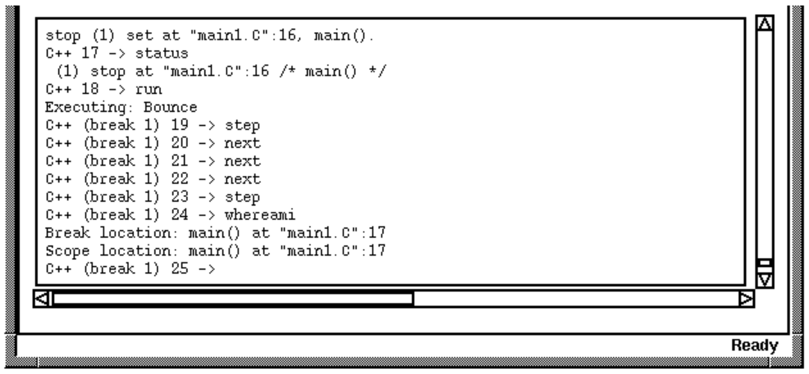
With this setting, the dialog box opens with the selected text as the default input line. This gives you an opportunity to see what text is selected and make any changes you want to make before you select **OK** or **Apply**.

For example, you can use the mouse to select the name of an identifier listed in the Source area, display the **Examine** menu, and select **Display**. When the Display dialog box opens, the selected text is in the input line.

For more information, see the **X resources** entry in the *ObjectCenter Reference*.

Entering ObjectCenter commands in the Workspace

When the Main Window opens, you are placed in an interactive work area, the Workspace, shown below. This provides a command-line interface to the CenterLine Engine.



```
stop (1) set at "main1.C":16, main().
C++ 17 -> status
 (1) stop at "main1.C":16 /* main() */
C++ 18 -> run
Executing: Bounce
C++ (break 1) 19 -> step
C++ (break 1) 20 -> next
C++ (break 1) 21 -> next
C++ (break 1) 22 -> next
C++ (break 1) 23 -> step
C++ (break 1) 24 -> whereami
Break location: main() at "main1.C":17
Scope location: main() at "main1.C":17
C++ (break 1) 25 ->
```

This interactive work area handles C++ and C statements as well as Workspace commands. The Workspace is also the output area for messages and results from C++ and C statements, shell commands, and some ObjectCenter commands.

How the debugging and language modes affect the Workspace

The CenterLine Engine has two debugging modes and two language modes, which affect the operation of the Workspace:

- Component debugging mode (cdm) is for debugging a project composed of source, object, and library file components.

  When you are in component debugging mode, the Workspace is either in C++ or C language mode. In C++ language mode, the Workspace uses the default prompt:

  ```
  C++ 1 ->
  ```

  When you are in C language mode, the Workspace uses the following prompt:

  ```
  C 1 ->
  ```

- Process debugging mode (pdm) is for debugging an externally linked executable alone or with a corefile or running process. When you are in process debugging mode, the Workspace uses the following prompt:

  ```
  pdm 1 ->
  ```

Entering C++ and C code

In either debugging mode, you can enter ObjectCenter commands and evaluate variables and C expressions. However, only component debugging mode (cdm) also offers the ObjectCenter interpreter, which allows you to use the Workspace to immediately execute any arbitrary C++ or C statement (depending on which language mode you are in). For information on entering C++ and C code in the Workspace, see 'Interactive prototyping and unit testing in the Workspace' on page 138.

Differences in Workspace usage

Throughout this manual, examples show the Workspace in component debugging mode. However, you can assume that most Workspace usages apply to both modes transparently. For information about differences when working in process debugging mode, see Chapter 6, “ Process debugging,” and the **pdm** entry in the *ObjectCenter Reference.*

Invoking ObjectCenter commands directly

In either mode, the Workspace functions as a command processor that allows you to enter ObjectCenter commands directly and to pass shell commands on to a subshell.

In general, the same ObjectCenter functionality is available either by using menus, buttons, and other graphical elements of the GUI or by entering ObjectCenter commands directly in the Workspace. You choose whichever means of access is most convenient for you at the moment.

Command format

Workspace commands take the following form:

command_name [*switches*] [*arguments*]

For example, you can use ObjectCenter's **whatis** command to display the use of a name. To see the declaration for the loaded function **do_bounce()**, you can type:

```
-> whatis do_bounce
extern int do_bounce(); /* defined */
```

To list the file **bounce.c** in the Source area, you can type:

```
-> list bounce.c
Listing file 'bounce.c', line 1 ...
```

NOTE To cancel a Workspace entry, press Control-c.

Identifying class members

You can use the same syntax to identify members of classes. For example, to see the declaration of the member function **getText()** in class **String**, you could type:

```
-> whatis getText
void String::getText() /* defined */
```

To list the definition of the member function **getLength()** in the class **String**, you could type:

```
-> list getLength
Listing file 'String.C', line 75 ...
```

Using the scoping operator

You can also use the scoping operator (::) when identifying members of classes. For example, you could also type the previous two examples as follows:

```
-> whatis String::getText
void String::getText() /* defined */
-> list String::getLength
```

Specifying operator functions

You can specify operator functions in the same way:

```
-> whatis String::operator=
String &String::operator =(String &s) /* defined */
```

If the argument is overloaded

When an ObjectCenter command is given a function as an argument and that function is overloaded, ObjectCenter lists all instances of the function and asks you which one you want to act on.

For example, the following statement uses the **stop** command to define a breakpoint in the constructors for the **String** class. Because there are several **String** constructors, ObjectCenter responds with a list of them:

```
-> stop in String::String
Name is overloaded:
(1) String::String(char *s)
(2) String::String(int l)
(3) String::String()
(4) String::String(String &str1)
#/quit/all [1] ?
```

Type the function's number to act on it. Type **a** to act on each instance of the function; in this case, a breakpoint would be set on each **String()** function. Type **q** to return to the Workspace without acting on the function.

You can specify an overloaded function's argument list (**signature**) to avoid having to choose from a list:

```
-> stop in String::String(char *)
stop (1) set at "String.C":13, String::String().
```

Getting information on ObjectCenter commands

This manual focuses on accomplishing your development tasks using the graphical elements of the GUI. At times, Workspace equivalents are shown as well.

For example, we usually suggest this way of opening the Data Browser:

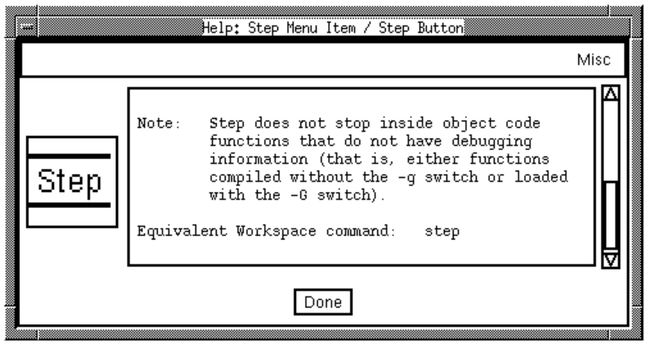
Open the **Examine** menu and select **Display**

We sometimes also add:

You can also enter the following in the Workspace:

```
-> display expression
```

ObjectCenter's spot help also shows Workspace equivalents. For example:



Note: Step does not stop inside object code functions that do not have debugging information (that is, either functions compiled without the -g switch or loaded with the -G switch).

Equivalent Workspace command: step

For a complete list of all the commands that you can enter in the Workspace, see the **commands** entry in the *ObjectCenter Reference*. Each command also has a separate *Reference* entry.

To get summary help on usage for a specific command right in the Workspace, you can use the **help** command:

```
-> help list
list - display source code lines
list line - display from line
list line line - display between lines
list func - display from top of function
list file - display from top of file
list "file":line - display specified line
list identifier - display defining location
list -n - display n lines before current location
```

Executing shell commands

ObjectCenter provides two Workspace commands to execute subshells from the Workspace:

- The **sh** command executes a Bourne subshell (**/bin/sh**).
- The **shell** command executes your default shell.

  The default shell is specified by ObjectCenter's **shell** option, which is set to the value of your UNIX **SHELL** environment variable when you start ObjectCenter.

Any command-line arguments that you give these commands are passed on to the subshell, and results from the shell command are displayed below the command line:

```
-> sh grep "test" main.c
test(1,2,3,4);
-> shell touch {term,last}.o
->
```

Using aliases for ObjectCenter commands

At startup, ObjectCenter automatically creates two aliases for the commonly used shell commands **pwd** and **ls**:

```
-> pwd
/usr/steinway
-> ls
a.c b.c c.c d.c
```

To see all the aliases currently defined, use the **alias** command with no arguments:

```
-> alias
ls              sh ls
pwd             sh pwd
assign          print
set             print
undisplay       sh echo "Use the 'delete' command to
                              remove display items."
restore         sh echo "Use the 'load' command to
                    restore project and image files."
```

To provide similar shortcuts for the commands you use most often in the Workspace, you can create your own aliases using the **alias** command. For example:

```
-> alias s step
```

You can use your local **.ocenterinit** or **.pdminit** file to define aliases across ObjectCenter sessions. Because ObjectCenter first looks in the current working directory for these local files, you can use different ones for each project as long as each project is in a different directory. You can also use the **ocenterinit** file to set system-wide options for component debugging mode. ObjectCenter reads the system-wide **ocenterinit** before **.ocenterinit**, so local aliases override any corresponding system aliases.

For more information, see the **alias** and **objectcenter** entries in the *ObjectCenter Reference*.

Other operations in the Workspace

The Workspace supports a range of extended input functionality, such as command history, name completion, and inline editing. This input functionality is based on similar features found in the **tcsh** shell (an extended version of the **csh** shell) and the **emacs** editor.

In addition to these extended input capabilities, ObjectCenter also offers functionality for general Workspace maintenance through clearing the Workspace, saving a session log, and displaying and saving your input history.

Clearing the Workspace

You can clear the Workspace by using the **clear** command from the Workspace pop-up menu. This places the Workspace prompt at the top of the pane. Clearing the Workspace pane does not affect any loaded files or attached libraries.

Saving a session log in component debugging mode

At any point, you can save to a file a transcript of Workspace actions. In this file, ObjectCenter saves all the Workspace lines specified in your resource file. The Workspace will save only a maximum number of lines. You can change the maximum number of lines, however, or make it unbounded. For information about changing the maximum number of lines, see the **X resources** entry in the *ObjectCenter Reference*.

To save a transcript, display the Workspace pop-up menu and select **Save Session to**. This displays a submenu.

You can use the default name (**~/ocenter.script**) or specify another name for your logfile.

Displaying your input history

You can display your previous input in the Workspace by using the scrollbar. You can also use ObjectCenter's **history** command to display previous input:

```
-> int i;
-> double d;
-> char c;
-> history
   1: int i;
   2: double d;
   3: char c;
   4: history
```

Saving your input history

ObjectCenter saves all input made in the Workspace in a temporary logfile that is deleted at the end of a session. The name of the logfile is specified by ObjectCenter's **logfile** option. You can specify that ObjectCenter keep a permanent logfile by using the **-f** command-line switch when starting ObjectCenter. For more information, see the **objectcenter** entry in the *ObjectCenter Reference*.

If you did not use the **-f** switch when starting ObjectCenter, you can still save the contents of the logfile at any point by redirecting the output of the **history** command:

```
-> history #> my_ocenter_log
```

The logfile records your Workspace input only; it does not show ObjectCenter's output. This logfile is affected by the line-number limit in the Workspace. You can change the line-number limit, however, or make it unbounded. For information about changing the line-number limit, see the **X resources** entry in the *ObjectCenter Reference*. You can also use the **edit workspace** command. See the **edit** entry in the *Reference* for more information.

More information on Workspace operations

For more details about Workspace input and maintenance features, see the **Workspace** entry in the *ObjectCenter Reference*.

Shortcuts for other operations in the Workspace

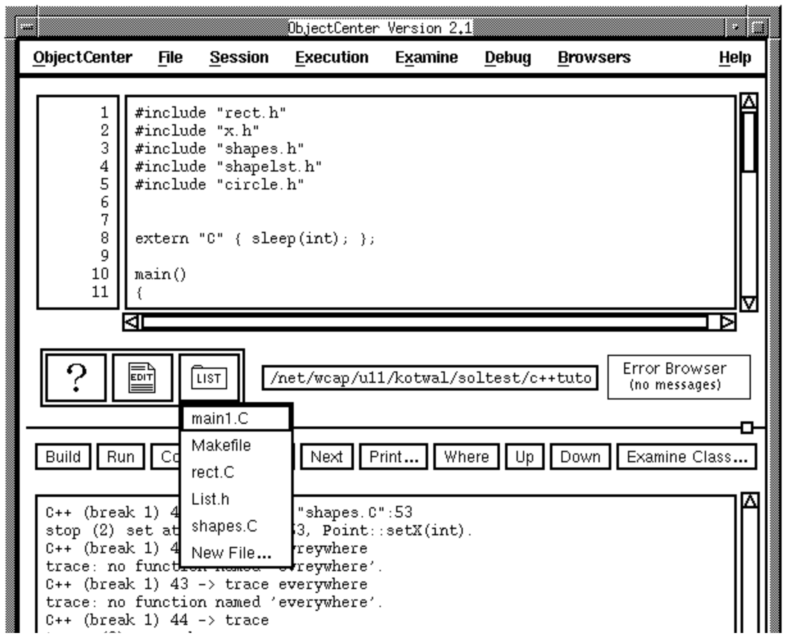
Beyond entering ObjectCenter commands at the prompt, you can use the Workspace shortcuts for operations on your command input as well as on the source code you have listed in the Source area: calling your editor on a specified location, getting information on listed expressions, and setting or deleting debugging items. Using these shortcuts, you can access these operations through pop-up menus and pop-up windows; see 'Main Window shortcuts with pop-up menus and windows' on page 31.

Listing source code

ObjectCenter lists source code in the Source area of the Main Window. Because it is such a fundamental aspect of all your programming tasks, ObjectCenter offers many ways for you to get your source code listed. And once you have it listed, you can use the Source area for many different operations on that source code: finding a particular place in the code, getting information on listed expressions, and setting or deleting debugging items such as breakpoints on specific lines of code.

Listing source code in the Source area

The method you use to list code depends on what window you are in and what means of access is the most convenient at a given moment. For example, the **List** menu in the Main Window allows you to choose from files you have listed previously, as shown in the following illustration.



```
#include "rect.h"
#include "x.h"
#include "shapes.h"
#include "shapelst.h"
#include "circle.h"


extern "C" { sleep(int); };

main()
{
```

```
/net/wcap/u11/kotwal/soltest/c++tuto
```

```
C++ (break 1) 4            "shapes.C":53
stop (2) set at            3, Point::setX(int).
C++ (break 1) 4            reywhere
trace: no funct            reywhere'.
C++ (break 1) 43 -> trace everywhere
trace: no function named 'everywhere'.
C++ (break 1) 44 -> trace
```

The following table describes the ways to list a file:

Table 4 Listing a File: Methods According to Work Area

Work Area	Ways to List a File
Main Window	From the **File** menu, select **List**. From the **List** menu below the Source area, select a previously listed file or select **New File**, which opens a file selection dialog box.
Project Browser	From the **File** menu, select **List**. From the pop-up menu in the Files area, select **List**.
Error Browser	Select a warning or error message.
Cross-Reference Browser	From the **Examine** menu, select **List**.
Inheritance Browser	Select **list**.
Class Examiner	Select **list**.
Workspace	Use the **list** command in the **Expressions Options** menu by pressing the Shift key and the Right mouse button. See 'Expressions Options (shift-right) menu' on page 206.

Shortcuts for other operations in the Source area

Beyond listing source code, you use the Source area for other operations on the source code you have listed: calling your editor on a specified location, getting information on listed expressions, and setting or deleting debugging items. You can access these operations through pop-up menus and pop-up windows; see 'Main Window shortcuts with pop-up menus and windows' on page 31.

Editing source code

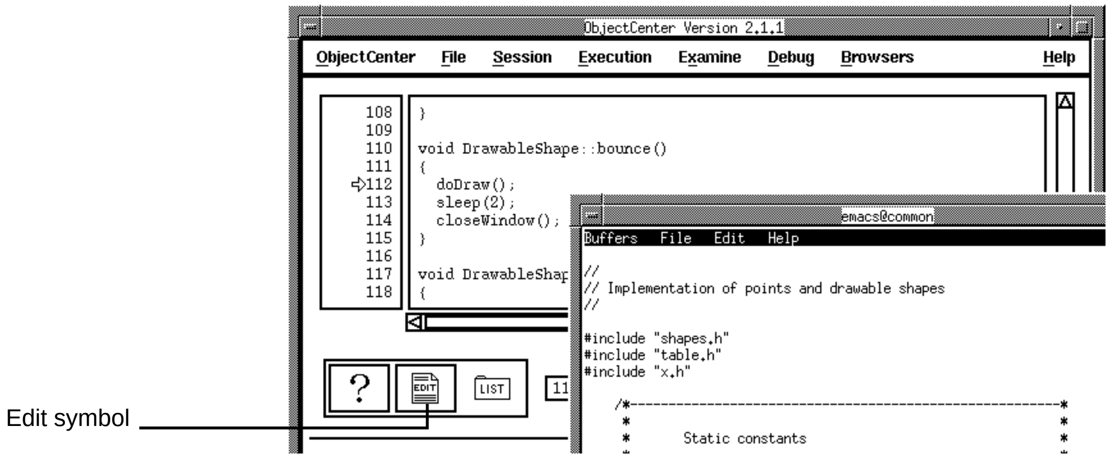
Through the Edit Server, ObjectCenter provides tight integration with your standard editor, either **vi** or GNU **emacs**. This tight integration allows you to get the source code you are concerned with at the moment into your editor quickly and easily.

For example, when you invoke your editor from a warning or error message in the Error Browser, ObjectCenter puts the source file containing that violation into your editor and places the cursor at the beginning of the line containing the problem.

You can also start ObjectCenter within an **emacs** session, which lets you edit your code directly in the source area. See the emacs integration entry in the *ObjectCenter Reference.*

Invoking your editor

Like listing files, editing your code is such a fundamental aspect of all your programming tasks that ObjectCenter provides many ways to do this. The method you use at a given moment depends on what window you are in and how you are working in that window. For example, selecting the Edit symbol in the Main Window invokes your editor on the currently listed file with the cursor at the top of the file, as shown in the following illustration:



```
}

void DrawableShape::bounce()
{
  doDraw();
  sleep(2);
  closeWindow();
}

void DrawableShap
{
```

```
//
// Implementation of points and drawable shapes
//

#include "shapes.h"
#include "table.h"
#include "x.h"

    /*-----------------------------------------------------*
     *                                                     *
     *      Static constants                               *
```

Table 5 lists the ways to invoke your editor:

Table 5 Invoking Your Editor: Methods According to Work Area

Work Area	Ways to Invoke Your Editor
Main Window	From the **File** menu, select **Edit**.
	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**.
Project Browser	From the **File** menu, select **Edit**.
	From the pop-up menu in the **Files** area, select **Edit**.
Error Browser	Select the Edit symbol at the left of a warning or error message.
Cross-Reference Browser	From the **Examine** menu, select **Edit**.
Inheritance Browser	Select **edit**.
Class Examiner	Select **edit**.
Workspace	Use the **edit** command in the **Expressions Options** menu by pressing the Shift key and the Right mouse button. See 'Expressions Options (shift-right) menu' on page 206.

Specifying your editor

As shipped, ObjectCenter provides integration for both the **vi** and GNU **emacs** editors. ObjectCenter uses the **vi** editor by default. To use an editor other than **vi**, specify this editor as the value of the **EDITOR** environment variable before you start up ObjectCenter. For example, at the shell type:

```
% setenv EDITOR emacs
% objectcenter
```

You cannot change editors within an ObjectCenter session. For more information see 'Connecting your editor to ObjectCenter' on page 226.

Quitting ObjectCenter

To exit an ObjectCenter session, from the **ObjectCenter** menu in the Main Window, select **Quit ObjectCenter**. In the dialog box, select **Quit** to exit.

This dialog box also gives you the choice of saving a *project file*. The next time you start ObjectCenter, you can quickly restore the state of your work by loading the project file.

For more information about using project files, see 'Loading an existing project file' on page 69.

You can also enter the following in the Workspace:

```
-> quit
```

To quit without being asked for confirmation, enter the following:

```
-> quit force
```

For more information about quitting ObjectCenter, see the **quit** entry in the *ObjectCenter Reference*.

Chapter 3 Managing your code in ObjectCenter

This chapter describes how to get your code into ObjectCenter and manage it effectively. It covers the following topics:

- *Your range of choices for working on code in ObjectCenter*
- *Deciding your basic approach*
- *Deciding which form to use for your components*
- *Loading individual components*
- *Loading components as a project*
- *Loading an existing project*
- *Establishing a project*
- *Managing individual components in your project*
- *Managing your whole project*
- *Enhancing performance for large projects*
- *Specifying an executable target*

Overview

Before you can use any of ObjectCenter's capabilities for interactive debugging or code comprehension, your first step is to get your code into the ObjectCenter programming environment.

In process debugging mode, you bring your code into ObjectCenter by specifying a fully linked executable as a debugging target.

In component debugging mode, you bring your code into ObjectCenter as source file, object file, and library components that together make up your current ObjectCenter project. As its incremental loader/linker brings these components in, ObjectCenter processes the files to obtain the information it needs for automatic error checking, debugging, and code visualization.

In addition to loading components for your project, when you are in component debugging mode, you can also set up and manage your project using the Project Browser and various ObjectCenter commands, options, and properties that operate either on individual files or on the project as a whole.

Your range of choices for bringing your code into ObjectCenter

Because ObjectCenter's functionality for interactive programming is so extensive, ObjectCenter gives you a wide range of choices in how you bring your code in and set up your work session. Figure 2 lays out the choices you can make:

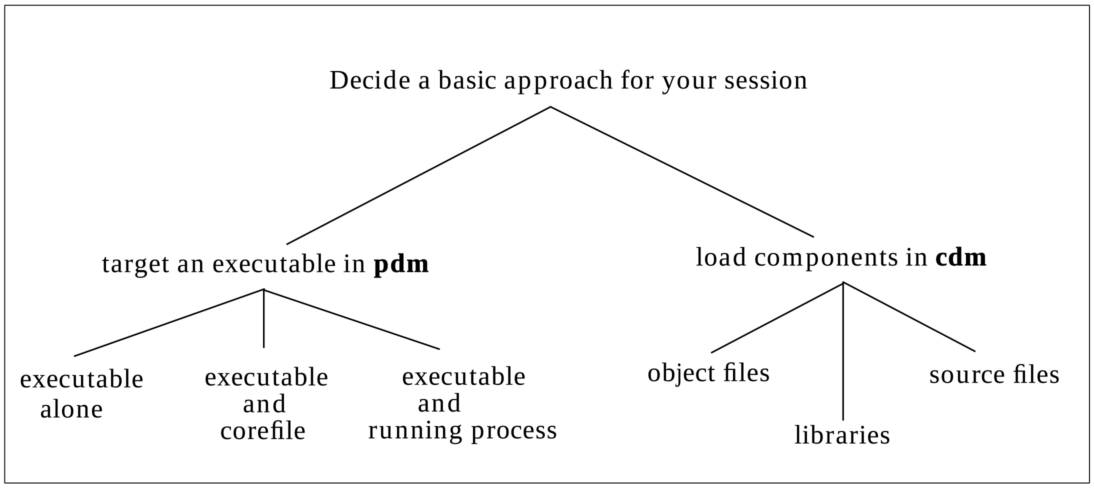

Decide a basic approach for your session
target an executable in **pdm**
load components in **cdm**
executable alone
executable and corefile
executable and running process
object files
libraries
source files

Figure 2 Ways You Can Bring Code In and Work On It in ObjectCenter

The following discussion describes each possibility.

Deciding your basic approach

The first choice in getting your code into ObjectCenter is between specifying a fully linked executable or loading source, object, and library files as separate components that make an ObjectCenter project.

Comparing performance factors

You decide on your basic approach based on the particular balance of performance factors that are most important to you for a given ObjectCenter session. Table 6 can help you decide.

Table 6 Performance Factors According to Debugging Mode

Performance Factor	**Component Debugging Mode**	**Process Debugging Mode**
Error checking	Full load-time error checking in source and run-time error checking in instrumented object and source.	Catches segmentation faults.
Interactive debugging capabilities	Full. Standard debugging actions. Interactive prototyping, using code fragments, and unit testing with the interpreter.	Some. Standard **gdb** debugging actions and debugging on machine instructions. Immediately locating an error that causes a crash by specifying a corefile.
Code comprehension	Data Browser for data-level understanding and ability to change values of elements directly in the Data Browser. Cross-Reference Browser for function-level understanding. Project Browser Contents window for file-level understanding. Inheritance Browser for class-level understanding. Class Examiner for class-member-level understanding.	Data Browser for data-level understanding.
Speed of setup	Slower setup. Loading source is slowest.	The fastest way to get your code in and start working.
Amount of memory	More memory required.	Demands least amount of memory.
Speed of execution	Ranging from full speed in regular object code to much slower in source.	Full speed of the machine.

Why load source, object, and library components?

You choose to load source, object, and library components if

- You want the incremental loader / linker for incremental development with quick turnaround on your edit, load, link, run, and test programming cycle.
- You want a C++ interpreter and a C interpreter for interactive prototyping and unit testing.
- You want the full range of code comprehension facilities that the Browsers provide.
- You want automatic load-time and run-time error checking.
- You want the full range of debugging facilities.

However, you need to keep in mind that to load source, object, and library components, you must be willing to accept heavier memory demands, slower setup time, and slower execution speed than with an executable.

Why target an executable?

You choose to target an executable if

- You want the fastest startup for getting to a specific bug.
- You want the lowest memory usage.
- You want to use a corefile to analyze the state of your program at the point of a crash.
- You want debugging at the assembly level.
- You want to debug a process that is already running.

However, you need to keep in mind that to target an executable you must be willing to sacrifice automatic error checking, some other debugging facilities, function and file-level code comprehension, and the C++ or the C interpreter for interactive prototyping and unit testing.

Deciding the form to use for components of a project

If you have decided to load source and object components in component debugging mode, your next choice is which form to use for these files.

The most clear-cut choice is between loading source or object code. If you decide to load object code, you have the additional choice of loading it with or without debugging information. And once object code is loaded, you have the choice of leaving it as it is or instrumenting it (using the ObjectCenter **instrument** command to enable run-time error checking on it).

You will almost always want to use **demand-driven code generation**, an option of ObjectCenter that saves significant load time and is on by default. When it is on, ObjectCenter loads only the source code or object code your program uses. For example, if your program uses only one class in a class library file, ObjectCenter does not load the entire class library. ObjectCenter loads only the code for the one class library you used. You can turn off demand-driven code generation whenever you want to experiment with class libraries or other code in the ObjectCenter programming environment.

For more information, see the **demand-driven code generation** entry in the *ObjectCenter Reference*.

This means that a component can take any of the following forms:

- Source code
- Instrumented object code with debugging information (compiled with the **-g** switch and loaded without the ObjectCenter **-G** switch)
- Instrumented object code without debugging information (compiled without **-g** or loaded with the ObjectCenter **-G** switch)
- Regular object code with debugging information
- Regular object code without debugging information
- Source code or object code of any type that ObjectCenter loaded with demand-driven code generation on
- Code consolidated with **ld -r**

- A shared library
- A static library

Which form you use for a given file depends on performance features you want to optimize for that file and your overall programming strategy at a given point.

TIP: Using templates

If your program uses templates, which are a feature of the C++ language new in Release 3.0, you need to look at the **templates** entry in the *ObjectCenter Reference* to take advantage of all of ObjectCenter's features for using templates. If you are new to templates, you also may be interested in the *Reference*'s discussion of such topics as the concept of templates in the C++ language, their use, and common pitfalls. 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 ObjectCenter.

TIP: Avoiding explicit loads of template declaration or definition files

Do not load template declaration or definition files explicitly. ObjectCenter handles these files automatically. For more information, see the **templates** entry in the *ObjectCenter Reference.*

TIP: Loading template instantiation modules

Use the **tmpl_instantiate_obj** option to tell ObjectCenter whether to load template instantiation modules as source code or object code. By default, this option is set, which means ObjectCenter loads template instantiation modules as object code.

Comparing performance factors

The following tables present the advantages of each of type of component and its performance factors. Table 7 compares loading code with demand-driven code generation on and off.

Table 7 Performance Factors With Demand-Driven Code Generation On and Off

Performance Factor	**Demand-Driven Code Generation= on**	**Demand-Driven Code Generation= off**
Speed of setup 1 = fastest 5 = slowest	4	5
Amount of memory 1 = least 5 = most	4	5
Load-time error checking	3	3
Standard debugging actions	3	3
Code visualization	3	5

Table 8 compares source code and the different types of object code. For more information about the performance factors for each type of component, see the **debugging** entry in the *ObjectCenter Reference.*

Table 8 Performance Factors for Source and Object Components

Performance Factor	**Source Code**	**Instrumented Object Code with Debug Info**	**Instrumented Object Code without Debug Info**	**Regular Object Code with Debug Info**	**Regular Object Code without Debug Info**
Setup speed 1 = fastest 5 = slowest	5	4	3	2	1
Memory use 1 = least 5 = most	5	4	2	3	1
Execution speed 1 = fastest 3 = slowest	3	2[a]	2	1[a]	1
Load-time error checking	Full.	Minimal. [b]	None.	Minimal.[b]	None.
Run-time error checking 1 = most 4 = least	1	2	3	4 (Minimal) [c]	4 (Minimal) [c]
Standard debugging actions	Full.	Some restrictions.[d,e]	Limited.[d, f]	Some restrictions.[d,e]	Limited.[d, f]
Code visualization	Full.	Some restrictions.[g]	Limited.[h]	Some restrictions.[g]	Limited.[h]

a. Increased paging due to heavier memory usage might possibly degrade execution speed compared to object code without debugging information. You can solve this by increasing available memory.

b. Only checks consistency of declarations and definitions across modules.

c. Only in certain standard library functions such as **malloc()** and **strcpy()**.

d. No tracing; no **stepout** command; some forms of the **action** command have no effect.

e. Due to the way the linker handles debugging information for consolidated (**ld -r**) object files, using debugging items is not as reliable with these files as it is with other object files containing debugging information.

f. Breakpoints can be set only on functions.

g. No macro definitions. The quality of type information available for object code with debugging information depends on how complete the debugging information supplied by your compiler is. Code visualization for object code with debugging information has the following restrictions for examining some variables: no class or function template information; no information on type **const**; no information on protection level of class members; references are treated as pointers; class browsing only on class hierarchy. To work around these restrictions, load a header file with the appropriate declarations, or swap one of the object files to source form.

h. Only cross-referencing functions and getting definitions for global symbols.

Additional performance considerations

In addition to the general performance factors presented in Table 8, performance can also be affected by the use of templates, ObjectCenter's header file skipping, and ObjectCenter's demand-driven code generation.

Skipping header files when compiling object files

If you load object files that require compilation (they do not exist or are out-of-date with their source or header files), you can decrease the compilation time with precompiled header files, using ObjectCenter's facility for avoiding recompilation of common header files.This facility saves and reuses an image of the compiled code for header files that are used in common by modules in your program. To enable header file skipping, use the **+k** switch to **CC**. For more information, see the **precompiled header files** entry in the *ObjectCenter Reference*.

Demand-driven code generation

You can increase performance when loading source or object code if you keep demand-driven code generation on, which is the default behavior. If the demand-driven code generation option is:

- On, ObjectCenter generates only the code your program uses. For example, if you use only one class in a class library, ObjectCenter generates only the code for the class you used. You will probably want to keep demand-driven code generation on almost all the time you are using ObjectCenter.
- Off, ObjectCenter generates all the code you have loaded, whether your program uses it or not. For example, even if you use only one class in a class library, ObjectCenter generates code for the entire library. You may want to turn demand-driven code generation off when you are experimenting with new code and with class libraries.

To turn demand-driven code generation off or on, use the **Enable Demand Driven Generation** option. See 'Setting project-wide properties' on page 85 and 'Setting properties for individual components' on page 80.

In a parallel fashion, you can also control demand-driven code printing for compilation from the shell by using the **-dd=on** switch. For more information see the "demand-driven code generation" entry in the *ObjectCenter Reference*.

Use the -G switch

You can save time and memory if you use the **-G** switch to load into ObjectCenter object files compiled with the **-g** switch. The **-g** switch of the compiler produces debug information, which you can choose to ignore in ObjectCenter if you **load** them into the environment with the **-G** switch.

Deciding an overall strategy

In addition to deciding how you want a particular file loaded in ObjectCenter, you need to decide the overall makeup of your project as a whole. As with the decision about the form for individual files, the particular mix of source and object files that you use for your project depends largely on your debugging strategy at a given point.

With a project composed of many components, you have a vast number of possible combinations. However, the most important configurations for a project fall into the following scenarios. Set up your project with one of the following combinations of code.

Source, instrumented object code, and regular object code

Load one or two files as source, a small group as instrumented object code with debugging information, and all the rest as regular object code without debugging information.

This is the recommended project configuration when you are developing code. You load a small group of files that are the focus of your active development as instrumented object code with debugging information. From this small group, you then swap in as source code the one or two files that you are actually making changes to at the moment. This gives you automatic error checking and full debugging capabilities on the new code as you add it. All other files are loaded as regular object code without debugging information.

Source and instrumented object code with debug info

Load one or two source files and the rest as instrumented object code with debugging information.

This is the recommended configuration when you are tracking down a bug. The one or two files you load or swap in as source code are the current focus of your debugging or development effort. For this focus area, you want the full range of ObjectCenter's interactive programming functionality available, including full load-time and run-time error checking. For the rest of your program, instrumented object code provides most run-time error checking, and loading object code with debugging information allows full interactive debugging in object files.

Instrumented object code with debug info

Load all files as instrumented object code with debugging information.

By loading no source code, you gain on setup and execution speeds. This is an good approach if you want to scan your program to localize a problem area, since instrumented object code gives most, but not all, run-time error checking capabilities. When you locate a trouble spot, you can swap the module in question to source code to have full error-checking and debugging capabilities.

Regular object code with debug info	Load all files as regular object code with debugging information. Using regular object code, rather than having ObjectCenter instrument it, increases performance for setup and execution times. Having debugging information gives you the full range of interactive debugging features.
Regular object code without debug info	Load all files as regular object code without debugging information. By dropping debugging information, you maximize setup and execution performance at the expense of many interactive debugging features. In this case, you may even want to consider using process debugging mode instead of component debugging mode. See Table 6 on page 53.
Source code only	Load all components as source files. This approach allows you to catch all load-time and run-time violations possible. Typically, you would use this approach as a final cleanup sweep through your entire program or to hunt down some otherwise intractable bugs. Loading all components as source files at the same time may not be possible depending on the size of your program and machine resources available.
Demand-driven code generation	Load all components with demand-driven code generation. By loading all components with demand-driven code generation, the ObjectCenter default, you save significant load time. When demand-driven code generation is on, ObjectCenter loads only the source code or object code your program uses. You can turn this option off when you want to experiment with new code.
Code fragments with other components	Load code fragments with other components. You can take advantage of ObjectCenter's interpreter by loading code fragments, such as those that include classes, to experiment with them. See the "Interactive prototyping" section on page 138 for more information.

For more information on these various possibilities for the makeup of your ObjectCenter project, see the “debugging” entry in the *ObjectCenter Reference.*

Choosing a method for loading the components

Once you have decided on which form to use for the different components of the project you want to work on, the next choice is the method to use for loading and setting up those components either as individual components or as a single project. The next sections discuss these two possibilities.

Loading individual components

The most direct way to bring source, object, and library components into ObjectCenter is to load each file or library individually. If you are not using templates, y ou can load any new file or library to an existing project or set up a project with a small number of files.

If you are using templates, you cannot explicitly load files containing template definitions or declarations. Instead, you must specify search paths for these files using the **-I** switch with the **sys_loadcxxflags** option (for system libraries and **#include** files) or the **load_flags** option (for user ones). See the **templates** entry in the *ObjectCenter Reference* to find out how to do so.

TIP: Using the -I (uppercase i) loading switch to specify directories for header files

To enable ObjectCenter to load your header files, be sure you use the **-I** (uppercase i) switch to specify the directories to search for header files. You do this with the **Load Flags** setting in the Properties window or the **load_flags** option. See 'Setting project-wide properties' on page 85.

Ways to load individual components directly

Table 9 lists all of the ways to load a source or object file without template definitions or declarations.

Table 9 Loading a Source or Object File: Methods According to Work Area

Work Area	**Ways to Load a File**
Main Window	From the **File** menu, select **Load**. This loads the file currently listed in the Source area.
Project Browser	From the **Project** menu, select **Add Files**. This opens a file selection dialog box. From the **File** menu, select **Load**.
Workspace	Use the **load** command.

Table 10 lists all of the ways to load a library.

Table 10 Loading a Library: Methods According to Work Area

Work Area	Ways to Load a File
Project Browser	From the **Project** menu, select **Add Libraries**. This opens a file selection dialog box. From the **File** menu, select **Load**. In the dialog box, use the **-l** switch format on the input line. For example, to load **libm.a**, enter **-lm**.
Workspace	Use the **load** command with the **-l** switch.

For more information, see the **load** entry in the *ObjectCenter Reference.*

What happens when you load a file or library

During the loading process, ObjectCenter's incremental loader / linker brings in your source or object file and gathers the information for ObjectCenter's interactive debugging and code comprehension capabilities. The speed of loading is directly related to the amount of information that ObjectCenter processes. Going in order from most information and slowest loading to least information and quickest loading, you can load files as source, object files with debugging information, or object files without debugging information.

If you instruct ObjectCenter to load a file that is already loaded, ObjectCenter checks to see if the file is up-to-date; if it is not up-to-date, ObjectCenter reloads it. A file may be out-of-date because a header file it includes is out-of-date.

By default, ObjectCenter instantiates templates later, when you issue a **link** or **build** command, rather than at load time. We do not recommend instantiating templates at load time. If you want to do so, however, see the "Switches for templates" section in the **templates** entry of the *ObjectCenter Reference.*

NOTE When loading a file containing object code with debugging information or source code, ObjectCenter performs load-time error checking. If it detects a serious problem that would prevent your code from compiling and linking, it automatically unloads the file. A message in the Workspace informs you that the file was unloaded, and the **Error Browser** button indicates that you have a new error message. For information on what to do, see 'What happens when load-time checking finds a static problem' on page 98.

Loading source code

If you request to load a source file and the corresponding object file is already loaded, ObjectCenter first unloads the object file. If the file is already loaded as source, then ObjectCenter processes the source code and translates it into an intermediate code. This intermediate code is used when you execute your program in ObjectCenter.

When translating C++ code, ObjectCenter does not expand inline functions (equivalent to using **+d** with **CC**). This is so you can debug inline functions.

Loading object code

If you request to load an object file and the corresponding source file is already loaded, ObjectCenter first unloads the source file. If the object file you want to load does not exist or is out-of-date, ObjectCenter causes a new object file to be compiled in one of two ways. If the directory containing the source file also contains a makefile, ObjectCenter does a **make** on the object file. If there is no makefile, ObjectCenter sends the source file to the compiler directly.

Once an up-to-date object file exists, ObjectCenter processes the object file to gather the information needed. The amount of information that ObjectCenter can gather depends on whether or not the object file contains debugging information.

For object files loaded with debugging information (compiled with the **-g** switch and loaded without the ObjectCenter **-G** switch), ObjectCenter has full type and source file information to enable most interactive debugging and code comprehension capabilities. (Macros are not accessible in object code.)

When determining whether an object file is up-to-date, ObjectCenter checks the object file to see which components your program actually used when creating the object file.

NOTE To enable ObjectCenter to determine which components your program actually used when creating the object files, you must compile your program with the **-g** switch (to include debug information). When you compile with the **-g** switch, the resulting object code lists the names of the header files your program used.

This behavior is different from the **make** program, which can determine header file dependencies even when you compiled your program without the **-g** switch.

TIP: Ensuring that header file dependencies are always checked

If ObjectCenter thinks a file is out-of-date but the **make** program does not, ObjectCenter believes **make**. This can occur when update dependencies are not included in your makefile. To ensure that header file dependencies are always checked when using makefiles, you can put explicit dependencies in your makefile. For example:

```
    x.o: x.h
```

For object files loaded without debugging information (compiled without the **-g** switch or loaded with the ObjectCenter **-G** switch), ObjectCenter only has information on text and data symbol definitions. This means that debugging capabilities are limited to setting breakpoints in a function, and code comprehension is limited to cross-referencing functions and finding the definitions for global symbols.

After you have loaded object code, either with or without debugging information, you can go one step further by instrumenting it to enable run-time error checking on it. See 'Instrumenting object files' on page 78.

Loading a library

At startup, ObjectCenter automatically loads the standard C++ library (**libC.a**) and the C library (**libc.a**) and any libraries that these libraries depend on. (The exact list of these other libraries depends on your platform.) On architectures that support shared libraries, ObjectCenter loads the shared version of the standard library. You can also load libraries directly in the same way that you load source and object files.

When you load a library, ObjectCenter attaches the library and makes the symbols it defines available for resolution of references from other modules in your project. ObjectCenter processes the contents of the library only to the extent of finding out what external symbols are contained there; it does not gather information for interactive debugging or code comprehension on library modules. (This is done only when you explicitly link your project, as explained next.)

Loading properties

When ObjectCenter loads a file or library, it uses the current values for the loading properties set in the Project-wide Properties window. For more information, see 'Setting project-wide properties' on page 85.

Automatic linking

At load time, ObjectCenter's incremental loader / linker automatically links source and object components as they are loaded except for template instantiations. ObjectCenter does not instantiate templates at load time. To instantiate templates, issue a **link** or **build** command.

When you reload a component or load a new component other than a template instantiation, the loader / linker only has to relink for the component being loaded, not for all the components that are already loaded. This linking resolves all the intermodule symbolic references among source and object files. It does not, however, resolve symbolic references to libraries.

TIP: Dealing with unresolved references to symbols in libraries

To deal with unresolved references to symbols in libraries, you need to explicitly link your project (see 'Linking your project' on page 82). You can use the **whatis** or **whereis** command to identify the location of the symbol. You can also use the **nm** command to find the library or module and make sure it is loaded.

With static libraries, ObjectCenter loads in the necessary modules from the libraries at the time when you link your project. For shared libraries, ObjectCenter loads in the necessary modules from the libraries only as they are needed when you run your program in ObjectCenter. Linking if you use templates

If you use templates, you need to explicitly link your project to uncover unresolved references you need to fix. See "Linking your project" on page 82.

When ObjectCenter links and loads in a library module, in addition to resolving symbols, ObjectCenter processes the module for information to enable interactive debugging and code comprehension.

More information

For more information about loading individual files and libraries, see the **load** entry in the *ObjectCenter Reference*.

Loading components as a project

In addition to loading individual files directly, ObjectCenter also provides several methods for you to load all your project components in a single operation. Figure 3 shows the range of choices available:

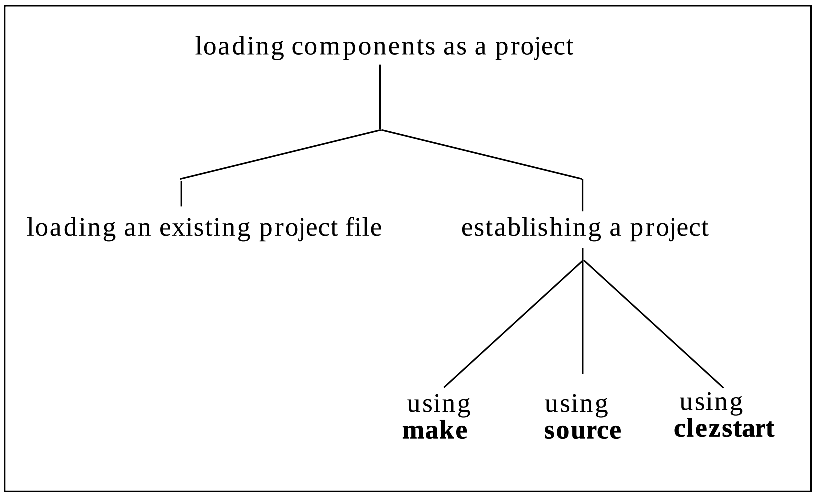


Figure 3 Methods for Loading Components as a Project

If you load your components as a single project, your first choice is between loading an existing ObjectCenter project or establishing a new project in your current session. You can load an existing project if you have previously saved your project as a project file. If you do not have an existing project saved as a project file, you can establish a new project for your session by issuing the ObjectCenter **make** command in the Workspace with a makefile containing special CL (CenterLine) targets, issuing the ObjectCenter **source** command in the Workspace with an ObjectCenter command file, or issuing the ObjectCenter **clezstart** command at the shell before issuing the **make** command. The next two sections discuss loading a project with and without an existing ObjectCenter project file.

Loading an existing project file

ObjectCenter allows you to save your current project as a project file. Once you have an existing project saved, you can load it simply by loading the project file.

Using a project file

At any point while working in ObjectCenter, you can save your current project as a project file. By loading this project file at a later time, you can easily re-establish your previous project and resume your work where you left off. The project file captures the following project setup features:

- Source, object, and library files that make up your project

  For each file in your project, your project file records the code form and loading flags for that file. The code form can be source or object code and, if object code, instrumented or regular. Loading flags are those like **-G** for object files loaded without debugging information.
- All current debugging items that are set
- Any warning numbers you have suppressed
- Settings for ObjectCenter options
- Settings for signals that are caught or ignored

Saving a project file

To save a project, in the Project Browser you display the **Project** menu and select **Save Project**. In the dialog box, you select **Save as Project File** and supply a name for the project file you want to save your current project in. The default name for a project file is **ocenter.proj**, but you can use any name you choose for your project file.

Alternatively, in the Main Window, you can use the **ObjectCenter** menu and select **Save Project**. You can also save a project file during the exit procedures when you quit ObjectCenter.

You can also enter the following in the Workspace:

```
-> save project name_of_your_project
```

If you type only **save project**, your project is saved under the name **ocenter.proj**.

Loading a project file

If you have already set up an ObjectCenter project, you can easily load the entire project into ObjectCenter by loading the project file for that project. To load a project file, display the **Project** menu and select **Load Project**. In the dialog box, supply the name of the project file. In the dialog box, also change the directory if the project file is not in your working directory, which appears by default.

You can also enter the following in the Workspace:

```
-> load project_file
```

When you load a project file, ObjectCenter loads the most recent versions of the source and object files in your project.

Replacing your present project when you load a project file

Loading a project file does not unload files previously loaded. Files in the project file are added to the previous files. If you want the project file to replace your previous files, you need to unload all your current files (see 'Unloading all files and libraries' on page 79).

Establishing a project

This section discusses the three ways in which you can load a project into ObjectCenter without having had to save the project previously in ObjectCenter. These are the three ways:

- Issue the ObjectCenter **make** command in the Workspace with a makefile containing special CL (CenterLine) targets.
- Issue the ObjectCenter **source** command in the Workspace with an ObjectCenter command file.
- Issue the ObjectCenter **clezstart** command at the shell to invoke the EZSTART utility that makes a copy of your original makefile and changes your existing targets to CL targets in the copy. You then issue the ObjectCenter **make** command on the copy to load the project into ObjectCenter.

You can use the ObjectCenter **make** or **source** commands to load your project at the start of each session, or you can load it as an ObjectCenter project file. No matter which way you load your project, you can always choose to save it in ObjectCenter as a project file.

Issuing the make command with CL targets

If you are familiar with maintaining a makefile and using the UNIX **make** utility, you may find the ObjectCenter **make** command the most convenient way to establish your project at the beginning of each new ObjectCenter session.

The ObjectCenter **make** command works with your own UNIX **make** utility and allows you to use makefiles to set up and manage your projects from within ObjectCenter. To do this, you simply add CL (CenterLine) targets to your makefiles.

A standard UNIX makefile target is composed of **shell lines** containing shell commands. For example:

```
a_standard_target: a.o b.o
   echo "starting a standard target"
   $(CC) $(CFLAGS) a.o b.o
```

In contrast, while a CL target can have shell lines like a standard target, it also has one or more CL lines. The syntax for a CL line is the following:

```
<tab>#ObjectCenter command
```

The following is an example of a CL target with a shell line and two CL lines:

```
a_cl_target: a.o b.o
   echo "starting a cl_target"
   #load a.o b.o
   #instrument a.o b.o
```

By adding this target to your makefile, you would then be able to issue the following command from the Workspace, using the same command-line arguments that you use with your standard UNIX **make** utility:

```
-> make a_cl_target
```

Invoking ObjectCenter's **make** command on this CL target would first echo the string at the shell, then load the object files **a.o** and **b.o** into ObjectCenter and instrument them.

In a similar way, you can add CL lines that use the **setopt** command to set ObjectCenter options at the same time you load your files. For example:

```
a_cl_target: a.o b.o
   #load a.o b.o
   #setopt program_name foo
```

For information on setting options, see 'Using ObjectCenter options' on page 216.

For more information on the ObjectCenter **make** command and customizing your makefiles with CL targets, see the **make** entry in the *ObjectCenter Reference*.

Using the source command with a command file

If you have an ObjectCenter command file that contains the ObjectCenter commands for setting up an ObjectCenter project, you set up your project by using the **source** Workspace command. For example, if your command file is named **my_commands**, you would issue the following command in the Workspace:

```
-> source my_commands
```

A command file is an ASCII file containing only ObjectCenter Workspace commands, with each command on a line by itself. For example, a simple command file might contain the following lines:

```
load *.o
instrument all
swap problem_child.o
```

To help you to capture the exact sequence of setup commands for a command file, you can start a new ObjectCenter session and use the exact series of Workspace commands you want recorded in the command file. When you have completed all the setup commands you want to capture, redirect the output of the **history** command to the file you will use for a command file. For example:

```
-> history #> my_commands
```

This file contains all the Workspace commands you have used. You can then edit this file to make any additional changes.

For more information, see the **source** entry in the *ObjectCenter Reference*.

Once you have established an ObjectCenter project for your current session, you use the Project Browser to manage your project during a session and from one session to the next.

Using clezstart

If you have an existing application for which you have a makefile with only standard targets (no CL targets) and you want to get all the files in this application loaded in as an ObjectCenter project, you can use the ObjectCenter EZSTART utility. The EZSTART utility allows you to import your application into ObjectCenter by using your standard makefile to generate a new makefile containing the appropriate CL targets. EZSTART names this new makefile **Makefile.cline**.

With this new makefile, you can use the ObjectCenter **make** command to load your files as a project. Once you have these files loaded in ObjectCenter, you can set properties for these files, set various ObjectCenter options, and then save your session as a project file. Alternatively, you can customize the CL targets in **Makefile.cline**, integrate them into your regular makefile, and use the **make** command to establish your project at the beginning of each session (see 'Issuing the make command with CL targets' on page 71).

How EZSTART works

Leaving your original makefile unchanged, EZSTART monitors your existing makefile as it builds your current application once. Based on this build of each file in your application, EZSTART constructs the makefile **Makefile.cline**, which contains the equivalent CL targets. This means that you do not have to change your existing makefile to begin using the CenterLine software.

Doing a complete conversion of standard targets

The basic operation of EZSTART is to create CL targets that load all of the files associated with your standard targets. You should use this method only if you:

- Use standard tools. By default, EZSTART recognizes the following tools only: **cc**, **clcc**, **CC**, **gcc**, **acc**, **ld**, **make**, **ar**, **mv**, and **cp**.
- Invoke these tools in your makefile without using an absolute path specification.

If your situation does not fit these conditions, see the **clezstart** entry in the *ObjectCenter Reference*.

A project import scenario

If you want to create CL targets to load all of the files that go into an application, do the following:

1. Remove all the object files and executables that make up the targets you want to build. Many users put a **clean** target in the makefile to do this.
2. At the shell, execute **clezstart** from the directory in which you normally execute **make**. Give the arguments to **clezstart** that you normally give to **make**.

For example, if you would ordinarily use the UNIX command **make myProgram**, you would substitute the following command:

```
% clezstart myProgram
```

The **clezstart** command creates the file **Makefile.cline** in the directory from which you executed it.

If you made an executable target called **myProgram**, there will be two targets in **Makefile.cline**. One of them is called **myProgram_obj**, which loads all the object files and libraries that go into **myProgram**, and the other is called **myProgram_src**, which loads all the source files and libraries. If you want to load object files, you would then use the target name with the **_obj** suffix as an argument for the **make** command in the Workspace:

```
-> make -f Makefile.cline myProgram_obj
```

You can create a third target named **myProgram_obj_nodebug**, which corresponds to the **myProgram_obj** target, except that ObjectCenter generates **myProgram_obj_nodebug** with the **-G** switch; that is, excluding debug information. For **clezstart** to create this third type of target automatically, you must make a change in your **clezstart_init** file. You must change the **cl_nodebug_target=no** option to **cl_nodebug_target=yes**.

For more information on using the EZSTART utilities, see the **clezstart** entry in the *ObjectCenter Reference*.

Managing individual components in your project

Viewing the components in your project

To get an overview of the components that make up your current project, you use the Project Browser. The Project Browser also lists files that failed to load because they contained an error.

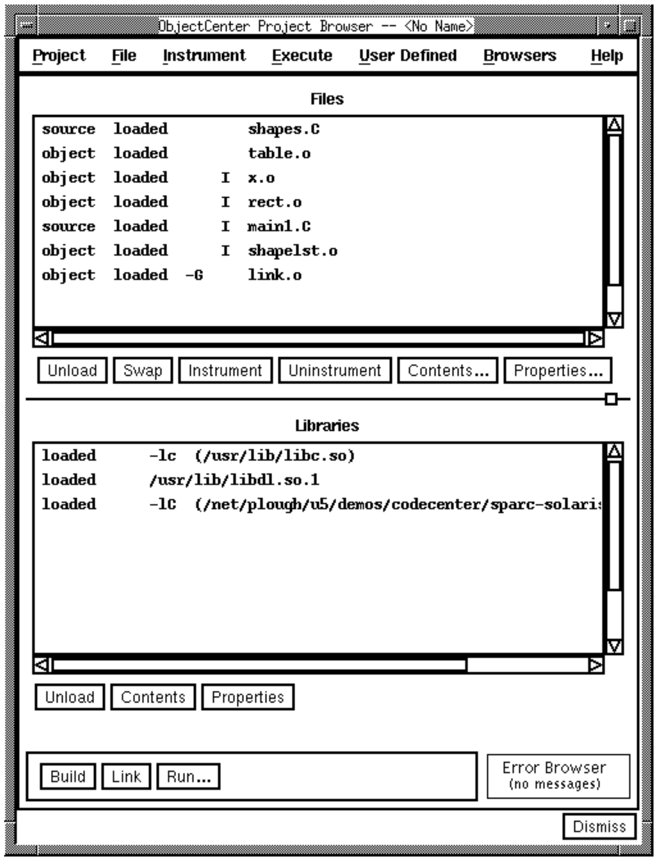
To open the Project Browser, in any primary window display the **Browsers** menu and select **Project Browser**.

You can also enter the following in the Workspace:

```
-> contents
```

The Project Browser has two functional areas for listing the components you have loaded: the Files area lists source and object files, and the Libraries area lists libraries.

The Project Browser uses an **I** to indicate that a file has the property for instrumenting object code set. (The file might be currently loaded in either object or source form.) The **-G** symbol indicates that an object file is loaded without debugging information. See the following illustration for an example.



```
source  loaded        shapes.C
object  loaded        table.o
object  loaded     I  x.o
object  loaded     I  rect.o
source  loaded     I  main1.C
object  loaded     I  shapelst.o
object  loaded  -G    link.o
```

```
loaded     -lc  (/usr/lib/libc.so)
loaded     /usr/lib/libdl.so.1
loaded     -lC  (/net/plough/u5/demos/codecenter/sparc-solari
```

Instrumenting object files

Enabling run-time error checking for object code in ObjectCenter is called *instrumenting* the object file. When you instrument a file, ObjectCenter gathers additional information on the file, which makes it possible for ObjectCenter to perform certain kinds of run-time error checking on the object code. For more information about memory usage and other aspects of instrumenting object files, see the **instrument** entry in the *ObjectCenter Reference*.

Loading files as instrumented object code

By default, files are loaded into ObjectCenter with the file property for instrumented object code unset. To have ObjectCenter load object files as instrumented code, set the **Instrument Object Files** property set in the Project-wide Properties window.

As with the other project-wide properties, the **Instrument Object Files** property in effect when you first load a file is remembered for that file throughout your ObjectCenter session, unless you explicitly change the properties in the File Properties window for that file.

Changing the project-wide property for instrumenting object files does not affect files that are currently loaded. For more information, see 'Setting project-wide properties' on page 85.

Changing from regular to instrumented object code

To change object files loaded as regular object files to instrumented object files, select the files in the Files area of the Project Browser and select the **Instrument** button. The files become instrumented. To instrument all object files, select **Instrument/Instrument All** from the Project Browser menu. All object files become instrumented. For more information see, 'Setting properties for individual components' on page 80.

You can also enter the following in the Workspace:

```
-> instrument filename.o
```

Once you instrument an object file, it remains instrumented until you explicitly change it to uninstrumented or unload it.

Changing from instrumented to regular object code

To change object files loaded as instrumented object files to regular object files, select the files in the Files area of the Project Browser and select the **Uninstrument** button. The files become uninstrumented. To uninstrument all object files, select **Instrument/Uninstrument All** from the Project Browser menu. All object files become uninstrumented. For more information see 'Setting properties for individual components' on page 80.

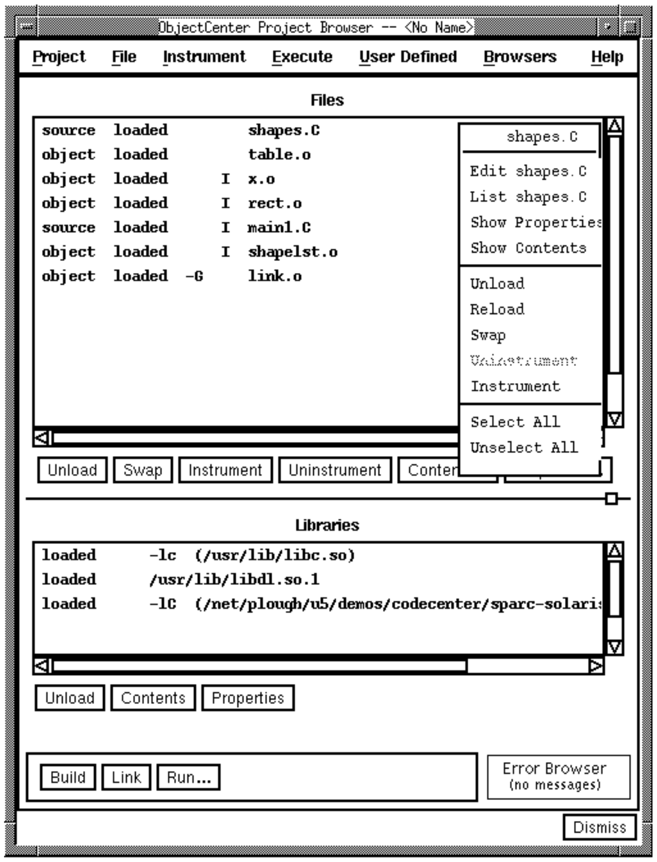
You can also enter the following in the Workspace:

```
-> uninstrument filename.o
```

Reloading files individually

Once you have a file loaded, you will want to reload the file any time you make a change to it using your editor. You keep your files up-to-date in ObjectCenter either by reloading files individually or by building your project (see 'Building your project' on page 82).

In the Files or Libraries areas of the Project Browser, move the mouse pointer over the line for the files you want to reload, display the pop-up menu, and select **Reload**.



```
source  loaded        shapes.C
object  loaded        table.o
object  loaded      I x.o
object  loaded      I rect.o
source  loaded      I main1.C
object  loaded      I shapelst.o
object  loaded   -G   link.o
```

```
loaded      -lc  (/usr/lib/libc.so)
loaded      /usr/lib/libdl.so.1
loaded      -lC  (/net/plough/u5/demos/codecenter/sparc-solari
```

When you issue a command to reload a source file, ObjectCenter reloads the source file only if the version currently loaded is out-of-date with the version on disk. For an object file, in addition to checking the loaded version of the file against the version on disk, ObjectCenter also checks against the corresponding source file on disk. If the object file is out-of-date in relation to the source file, ObjectCenter causes the file to be recompiled (by calling **make** if there is a makefile in the source directory or, if not, by sending the source file directly to the compiler) and then reloads the file.

When you reload a file, ObjectCenter retains the same properties for the file that it had before you reloaded it. For more information, see 'Setting properties for individual components' on page 80 and 'Setting project-wide properties' on page 85.

Unloading files

To remove a file from your current project, you unload it. When you unload a file, you lose any property settings for that file. If you later load the file again, you need to reset these properties (see 'Setting properties for individual components' on page 80). When you unload a file, there is no need to relink the remaining files.

You can unload a file or library by selecting **File/Unload** in the Main Window or Project Browser or by using the **unload** command in the Workspace.

Unloading all files and libraries

To unload all source and object files, with the mouse pointer in the Files area, display the pop-up menu and select **Select All**. Then select the **Unload** button in the Files area.

You can also enter the following in the Workspace:

```
-> unload user
```

To unload all libraries, with the mouse pointer in the Libraries area, display the pop-up menu and select **Select All**. Then select the **Unload** button in the Libraries area.

By combining the two methods just described, you can unload your entire project.

You can also enter the following in the Workspace:

```
-> unload all
```

For more information on unloading your files, see the **unload** entry in the *ObjectCenter Reference*.

Swapping files

At any time, you can change a file that you have loaded from object code to source or from source code to object by swapping it. When you swap a file, ObjectCenter first unloads the file in its current form (for example, **main.o**) and then loads the corresponding file (**main.c**).

TIP: Specifying the search path for swapping files

When you swap a file, by default ObjectCenter searches in the same directory where you loaded the file being swapped out. To specify the search path for swapping, use the **Search Path for Files** and **Use Search Path When Swapping** project-wide property settings (see 'Setting project-wide properties' on page 85).

Swapping is especially useful when you have a file in source form as the focus of your debugging effort and finish debugging it. You would then swap it to object form to get better performance. Likewise, when you have a file loaded as object code and decide to make that file the focus of your debugging effort, you would swap it to source code.

When you swap a file, ObjectCenter retains the same properties for the file that it had before you swapped it. For more information, see 'Setting properties for individual components' on page 80 and 'Setting project-wide properties' on page 85.

You can swap a file or library by selecting **File/Swap** in the Main Window or Project Browser or by using the **swap** command in the Workspace.

For more information, see the **swap** entry in the *ObjectCenter Reference*.

Setting properties for individual components

An important way to manage the individual components in your project is by setting properties on specific files and libraries. Once you set properties for a component, these properties stay in effect as you swap or reload the file or build your project. If you unload a file or library, any properties you have set for that component are lost and need to be reset if you later load that file or library again individually.

Setting properties for source or object files

To set properties on source or object files, go to the Files area of the Project Browser and select all the files for which you want to set properties. Then select the **Properties** button below the Files area. A File Properties window opens for each of the selected files. You can set the following properties:

- Load flags
- Source language (C, C++, or Other / Unknown)
- Whether to ignore warnings when loading a source file
- Whether to ignore debugging information
- Whether to instrument an object
- Whether to enable demand-driven code generation

Setting properties for libraries

To set properties on libraries, go to the Libraries area of the Project Browser and select all the libraries for which you want to set properties. Then select the **Properties** button below the Files area. A Library Properties window opens for each of the selected files. You can set the following properties:

- Load flags
- Whether to ignore warnings when loading
- Whether to load debugging information

Managing your whole project

Building your project

As you make changes to the source code for source and object files you have loaded in ObjectCenter, your project becomes out-of-date. To keep your project current with your source code, you need to build your project.

You can build your current project by selecting **Session/Build Project** in the Main Window, **Execute/Build Project** in the Project Browser, or by using the **build** command in the Workspace.

When you build your project, ObjectCenter checks the dependencies for all the source and object files that you have loaded. ObjectCenter reloads and relinks a source file if the file itself has changed or if it is out-of-date with the header files it depends on.

For more information on building a project, see the **build** entry in the *ObjectCenter Reference*.

Linking your project

ObjectCenter's incremental loader / linker automatically links together source and object modules as these are loaded, resolving symbolic references among these modules. This automatic linking does not, however, resolve symbolic references to libraries.

NOTE To resolve symbolic references to libraries and template definitions if you use templates, you need to explicitly link your project.

Ways to explicitly link your project

You can link your current project by selecting **Session/Link Project** in the Main Window, **Execute/Link Project** in the Project Browser, or by using the **link** command in the Workspace.

Dealing with unresolved symbols

If you select **Link** in the Project Browser or issue the **unres** command in the Workspace, the Project Browser opens an Unresolved Symbols window, listing all unresolved symbols. You can select one of the unresolved symbols and open the Cross-Reference Browser to see where the unresolved symbols are used. See the "Cross-Reference Browser" section on page 191.

If you use templates and you have unresolved references to template definitions, make sure:

- The template definitions and declarations are in files named according to the conventions in the “Coding conventions” section of the **templates** entry in the *ObjectCenter Reference*.
- The path to each template definition and declaration file is specified with the **-I** switch in **load_flags** or in **sys_loadcxxflags**. See the “Options” section of the **templates** entry in the *ObjectCenter Reference*.

To resolve unresolved references to definitions other than templates after linking, do the following:

1. Check the source and object files listed in the Files area of the Project Browser. If files are missing, load them individually.
2. If files listed in the Project Browser have a **failed** load status, then check the Error Browser to see if load-time errors are reported for these files. If so, correct the errors and reload the files.
3. Check the Libraries area of the Project Browser for missing libraries. If any libraries are missing, load them individually.
4. Link your project again to ensure there are no more unresolved symbols.

For more information on linking your project, see the **link** entry in the *ObjectCenter Reference*.

Running your project

ObjectCenter’s interpreter allows you to either run your whole project or execute any part of your code at any time.

You can run your current project by selecting the **Run** button in the Control Panel of the Main Window, **Execute/Run** in the Project Browser, or by using the **run** command in the Workspace.

When you run your project, ObjectCenter first links it and then executes **main()**. If ObjectCenter reports unresolved symbols as a result of attempting to link your project, you need to resolve these symbols. If you use templates and ObjectCenter reports unresolved symbols as a result of attempting to instantiate templates, you need to resolve these symbols also. See 'Linking your project' on page 82.

When you start ObjectCenter, your program opens a **clxterm** window to run in, and returns control to the shell in which you invoked ObjectCenter. A **clxterm** window is an **xterm** window running under ObjectCenter. You can set options in a **clxterm** window just as you can with an **xterm** window.

To avoid creating the **clxterm** window, called the Run Window, use the **-no_run_window** switch when you invoke ObjectCenter. The program's input and output will instead go to the shell in which you invoked ObjectCenter. When you use the **-no_run_window** switch, you are unable to interrupt ObjectCenter and unable to place it in the background. This switch is intended for debugging applications that need specific terminal support rather than a generic **xterm** window.

NOTE You should not start ObjectCenter in the background using the **-no_run_window** switch. Your program could have undesirable input/output behavior.

To create the Run Window, but avoid returning immediate control to the shell, use the **-no_fork** switch. When you use the **-no_fork** switch, control returns to the shell only when you enter the suspend character (usually Control-Z) or exit ObjectCenter. After you type the suspend character in the shell, you must type **bg** to enable your program to direct output again to the Run Window. If you do not use the **-no_fork** switch, control is immediately returned to the shell.

TIP: Clearing the Run Window

To clear the Run Window, place your cursor over it and hold down the Control key and the Middle mouse button. From the popup menu that appears, select "Reset and Clear Saved Lines".

The Run Window behaves like an xterm window. Each Control key/mouse button pair pops up a different menu. For more information, refer to the **xterm** UNIX manual page.

By default, issuing the **run** or **start** command will automatically restore the Run Window if it has been iconified. To prevent this default behavior, set the **win_no_raise** option in the Workspace or in your startup file.

TIP: Using a different terminal emulator for program input and output

If you want to use a different terminal emulator for input and output from ObjectCenter's default, start another ObjectCenter session from that terminal emulator with the **-no_run_window** switch.

For more information on running your project, see the **run** and **rerun** entries in the *ObjectCenter Reference.*

Setting project-wide properties

When loading a file, ObjectCenter uses the project-wide loading properties specified by the Project-wide Properties window. These properties are remembered for that file throughout your session as you swap or reload files or build your project. Alternatively, to have a file take on new project-wide properties, you can unload and then load the file.

TIP: Changing properties for a file that is already loaded

To change a property for a file that is currently loaded, do one of the following:

- Set new properties for that file in the File Properties window (See 'Setting properties for individual components' on page 80).
- Unload the file, set the desired project-wide properties, and then load the file again.
- Reload the file from the Workspace using the **load** command with explicit load flags. (These will replace the old project-wide load-flag properties for that file.)

To set these properties, go to the Project Browser, display the **Project** menu, and select **Project Properties**. In the Project-wide Properties window, you can specify the loading properties, described in Table 11, which correspond to the indicated ObjectCenter options.

Table 11 Project-Wide Properties Window

Project-Wide Property	**Description**	**Corresponding Option or Switch**
Program Name	Specifies the value of the first argument to **main()**, which is **argv[0]**.	**program_name**
Search Path for Files	Specifies the order for searching directories when any of the following ObjectCenter commands are invoked: **cd**, **edit**, **list**, **load**, or **swap** (if the **swap_uses_path** option is set). The **path** does not affect where ObjectCenter searches for header files. For this, use the **-I** switch with the Load Flags property or **load_flags** option.	**path**
Use Search Path When Swapping	Specifies whether to use the value of the **path** option when the **swap** command is invoked. (Use if selected.)	**swap_uses_path**
Load Flags	Specifies the default switches used for new files being loaded. This property applies only the first time a file is loaded and only if the **load** command is called without any switches specified explicitly. Changing the Load Flags properties does not affect any files currently loaded. Once you load a file (even if the load fails), unless you explicitly specify new switches when issuing the **load** command, ObjectCenter always reloads the file using the same load switches in effect when it was loaded originally. Be sure to use the **-I** switch to specify additional directories to search for header files. ObjectCenter always uses **-L** switches (switches for specifying the search path for libraries) with **load_flags**. If you want to change any of the properties explicitly listed elsewhere on the Project-Wide Properties window, do so where they are explicitly listed. See the entries in this table for the following items explicitly listed elsewhere on the Project-Wide Property window: Assume ANSI C, Ignore Warnings When Loading, Load Debugging Information, Instrument Object Files, Language Properties, Instantiate Templates as Object Code, Enable Demand Driven Generation.	**load_flags** (without the **-w** or **-G** switches)

Table 11 Project-Wide Properties Window (Continued)

Project-Wide Property	**Description**	**Corresponding Option or Switch**
Assume ANSI C	Specifies whether to conform to the ANSI C standard. (Conform if selected.)	**ansi**
Ignore Warnings When Loading	Specifies whether to automatically suppress all load-time warnings. (Suppress if selected.)	**load_flags** (with the **-w** switch)
Load Debugging Information	Specifies whether to load in debugging information contained in object files compiled outside ObjectCenter with the **-g** switch. (Use if selected.)	**load_flags** (with the **-G** switch)
Instrument Object Files	Specifies whether to automatically instrument each object file when it is loaded. (Instrument if selected.)	**instrument_all**
C Suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C source file that corresponds to a given object file.	**c_suffixes**
C++ Suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.	**cxx_suffixes**
Instantiate Templates as Object Code	If set, ObjectCenter instantiates templates as object code. If unset, ObjectCenter instantiates templates as source code.	t**mpl_instantiate_obj**
Enable Demand Driven Generation	Specifies complete demand-driven code generation. Only C++ code actually used in a source module you are loading (together with any included header files) is generated. This is the default setting.	l**oad -dd=on**

For more information on the options and switches related to these project-wide properties, see the **load** entry in the ObjectCenter *Reference*. For information on how to set options, see 'Using ObjectCenter options' on page 216.

TIP: Changing load flags for a file already loaded

Changing the Load Flags property or the **load_flags** option does not affect any currently loaded files. To change load flags for a file already loaded, use the **load** command in the Workspace and explicitly specify new load switches for it.

Alternatively, after you change values for the Load Flags property or the **load_flags** option, you can unload files to clear ObjectCenter memory of their original load flags and then reload them so they will pick up the new Load Flags settings.

Enhancing performance for large projects

When working with large ObjectCenter projects, you can use the techniques presented in this section to enhance performance. However, as already discussed in this chapter, increased performance in loading, executing, and memory conservation are all factors that need to be balanced against debugging capabilities available.

The following table compares various techniques and the areas of performance that they affect.

Table 12 Effects of Various Techniques for Enhancing Performance

	Performance Enhancement Gained		
Technique to Use	**Speed of Project Setup**	**Speed of Execution**	**Memory Conservation**
Consolidate object files	✓		✓
Load object, not source	✓	✓	✓
Do not load debugging information	✓	✓	✓
Use regular object code, not instrumented object code	✓	✓	✓
Set the **save_memory** option			✓

Consolidating object files

You can speed up the load process and conserve memory by combining many smaller object files into one large object file with the **ld -r** linker command. You can use this approach for object files that you are not changing much.

The loading time for the consolidated (**ld -r**) object file will be much faster than the total time for the separate smaller files.

NOTE Due to the way the linker handles debugging information for consolidated (**ld -r**) object files, using debugging items is not as reliable with these files as it is with other object files containing debugging information.

The specific switches you use with the **ld -r** linker command depends on your platform. For more information, see the UNIX manual page for the **ld** command on your system.

Loading object files rather than source files

Object files load faster, execute faster, and take less memory than source files. You can increase performance in all these areas by loading all the files of your project in object form, except for a few selected files that you load in source form in which you have maximum error checking, debugging, and code visualization capabilities. Or you can always load all files in object form and then swap a few selected files to source.

Not loading in debugging information

Object files loaded without debugging information load faster and use less memory. To increase performance in these areas with object files, either you can use object files that were compiled without debugging information (compiled without **-g**) or you can use the **-G** switch with the **load** command.

Using regular object code rather than instrumented code

While not part of the loading process itself, instrumenting object code adds to setup time. Instrumented object code also executes more slowly. To minimize setup time for a large project, only instrument object code that you want to examine more carefully during a given session.

Setting the save_memory option

Another way to conserve memory is to set the **save_memory** option (see 'Using ObjectCenter options' on page 216).

When the **save_memory** option is set, global variables and allocated data take up less memory. This is effective for memory conservation with applications that have large data structures, such as large arrays like **int a[500000]**.

You need to set the **save_memory** option before you load files. Setting this option reduces run-time violation checking capabilities. For more information on the **save_memory** option, see the **options** entry in the *ObjectCenter Reference*.

Keeping demand-driven code generation on

By keeping demand-driven code generation on, which is the ObjectCenter default, you can save significant load time. When demand-driven code generation is on, ObjectCenter loads only the source code or object code your program uses. For example, if your program uses only one class in a class library file, ObjectCenter does not load the entire class library. ObjectCenter loads only the code for the one class library you used.

Specifying an executable target

If you have decided to target an executable in process debugging mode, your next choice is which form of executable to target: an executable file alone or with a corefile or running process. This choice depends on your programming objective at a particular time.

Using an executable alone

If you want the simplest, most direct route to jumping into a debugging session, you specify an executable file. To specify an executable file as your debugging target, use the **debug** command and give the name of the file as an argument. For example:

```
pdm 2 -> debug bounce
Debugging program 'bounce'
```

Using an executable with a corefile

If your program has dumped core and you want to go immediately to the point of the crash, you specify a corefile for your executable. To specify a corefile with your executable, use the **debug** command and give the name of the executable and its associated corefile as arguments. For example:

```
-> debug bounce_dump core
Debugging program 'bounce_dump'
Core was generated by 'bounce_dump'.
Program terminated with signal 11, Segmentation
fault.
#0  0x10df0 in store_shape (count=0,
shape=0xeffff590 "rectangle") at shape.c:11
    11          *old = *new;
```

Using an executable with a running process

If you have a program that you need to debug as a running process, you use the **debug** command and give as arguments the name of the executable and the id number of the associated running process.

For example, with the program **bounces** already running outside of ObjectCenter with the process id **866**:

```
pdm 30 -> sh ps -auxww |grep bounces
petiprin 866 90.9 1.3 36 356 p0 R 11:39 1:06 bounces
petiprin 863 15.4 0.4 28 112 p2 S 11:38 0:00 sh -c
ps -auxww |grep bounces
```

You specify this process as a target in the following way:

```
pdm 31 -> debug bounces 866
Debugging program ‘bounces’ (previous program
‘/tmp_mnt/hosts/w+cap/u8/petiprin/ctutor_dir/bounces’
)
Resetting to top level.
pdm (break 1) 32 ->
```

More information

For more information on when you would use each type of executable target, see 'How you work in process debugging mode' on page 165.

Chapter 4 Fixing static errors

This chapter covers ObjectCenter's load-time error checking. It describes how ObjectCenter checks for static errors and how to use the Error Browser to deal with load-time errors and warnings. The chapter covers the following topics:

- *What static errors ObjectCenter finds*
- *What happens when ObjectCenter finds a static problem*
- *Using the Error Browser to deal with warnings and errors*

Overview

When you load a source file in component debugging mode, ObjectCenter automatically checks it for static errors; its load-time error checker functions like a super-lint utility. If ObjectCenter finds any static problems, it notifies you of these load-time violations (warnings or errors) as they are encountered.

Using the Error Browser, you can either correct these static problems as you go or suppress them to deal with later. ObjectCenter's incremental loader/linker and integration with your editor provide quick turnaround to allow you to incrementally fix static errors as you load the components of your project.

What static problems ObjectCenter finds

The kinds of static problems ObjectCenter finds

ObjectCenter's load-time checking finds static problems in the following categories:

- I/O errors
- Illegal characters
- Illegal constant formats
- Illegal escape sequences
- Lexical constant overflow
- Improper comments
- Preprocessing violations
- Macro expansion violations
- Syntax errors
- Illegal statements
- Illegal expressions
- Undefined identifiers
- Unused variables

- Improper type specifiers
- Declaration violations
- Initialization violations
- Redefinition violations
- Linking violations
- C++ ambiguity errors
- Class-specific errors
- Operator errors
- Virtual function errors

For a complete list of the individual violations that ObjectCenter reports for each category, see the **violations** entry in the online *Reference*.

Undetected function argument mismatches in C++ and C code

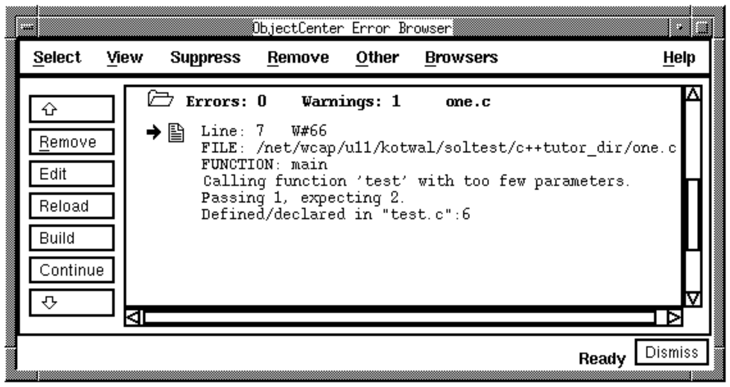
If you load C code that does not declare functions with prototypes, argument mismatches concerning the number and type of arguments for functions may go undetected depending on the order in which files are loaded.

This is because the function argument mismatch violations are not reported until the defining instance of the function has been seen. Therefore, all calls to the function that were loaded prior to the defining instance of the function are not checked properly.

For example, suppose that the **test()** function, which is defined in the **test.c** file, is called with one argument in file **one.c** and three arguments in the file **two.c**, as follows:

```
one.c:     test(1);
two.c:     test(1,2,3);
test.c:    test(i, j) int i, j;
```

If both **one.c** and **two.c** are loaded (with **-C**, since they are C files) prior to **test.c**, then no violations are reported. If however, you load **test.c** first, the Error Browser reports a warning in **one.c** and **two.c.** Here is the warning the Error Browser displays after loading the file **test.c** and then the file **one.c**.



```
Errors: 0    Warnings: 1     one.c

Line: 7   W#66
FILE: /net/wcap/u11/kotwal/soltest/c++tutor_dir/one.c
FUNCTION: main
Calling function 'test' with too few parameters.
Passing 1, expecting 2.
Defined/declared in "test.c":6
```

Since all function argument mismatches are detected at run time, this situation is not as potentially troublesome as it may appear.

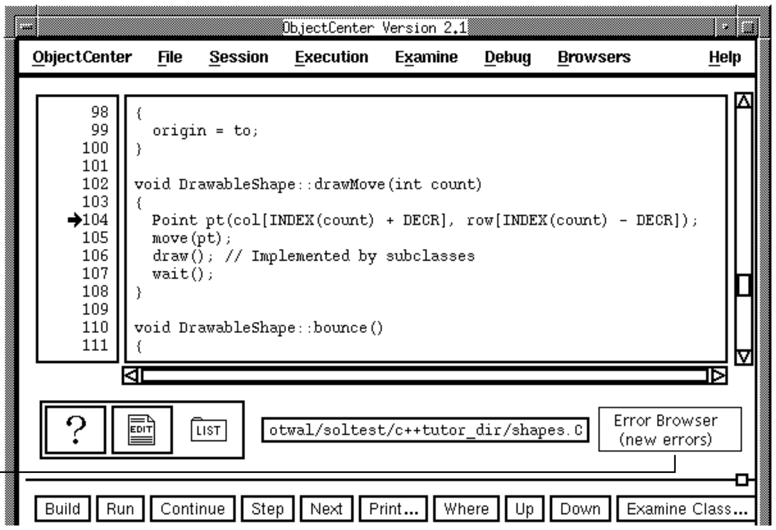
To find all of these warnings at load time, you must generate a prototype file. For more information, see the **proto** entry in the *ObjectCenter Reference*.

What happens when load-time checking finds a static problem

When ObjectCenter's load-time error checker finds a static problem, ObjectCenter notifies you with the **Error Browser** button and lists the violation message in the Error Browser. For load-time errors, you use ObjectCenter's incremental loader / linker and close integration with your editor to correct the source code and quickly load and link the corrections into your ObjectCenter project. For load-time warnings, you can either correct them as you go or suppress them and deal with them at a later time.

The Error Browser button

To notify you that new violation messages have arrived in the Error Browser, the Main Window and the Project Browser have an **Error Browser** button that announces new error and warning messages.



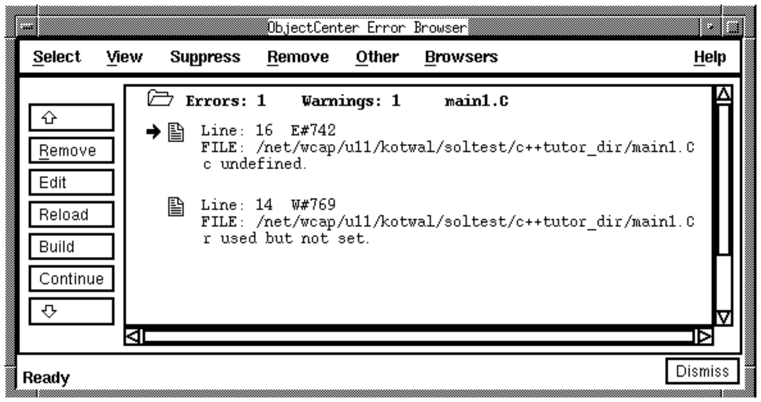
```
{
  origin = to;
}

void DrawableShape::drawMove(int count)
{
  Point pt(col[INDEX(count) + DECR], row[INDEX(count) - DECR]);
  move(pt);
  draw(); // Implemented by subclasses
  wait();
}

void DrawableShape::bounce()
{
```

otwal/soltest/c++tutor_dir/shapes.C

To deal with load-time violations, you can open the Error Browser by selecting the **Error Browser** button in the Button Panel of the Main Window or the Project Browser. You can also select **Error Browser** from the **Browsers** menu in any primary window.

The Error Browser

As ObjectCenter finds load-time violations in source files it is loading, it lists a message summarizing each violation in the Error Browser. Here's an illustration:



```
Errors: 1    Warnings: 1     main1.C

Line: 16  E#742
FILE: /net/wcap/u11/kotwal/soltest/c++tutor_dir/main1.C
c undefined.

Line: 14  W#769
FILE: /net/wcap/u11/kotwal/soltest/c++tutor_dir/main1.C
r used but not set.
```

The Error Browser divides the static problems it finds into two categories—more serious or less serious problems—and generates a different type of violation message for each kind of problem:

- Load-time error messages

  Load-time error messages indicate serious problems in your code that would prevent your code from compiling and linking.

  Because a file containing a load-time error cannot be linked with the rest of your ObjectCenter project, ObjectCenter automatically unloads the file. The Project Browser lists the load status of the file as **failed**. You cannot use the file containing the error inside ObjectCenter, and the definitions in this file are not available to your project. ObjectCenter, however, remembers the property settings for files that failed to load.

- Load-time warning messages

  Load-time warning messages indicate static problems that are less serious. These problems would not prevent your code from compiling or linking. However, they indicate potential trouble spots in your code and, at some point, need to be attended to in order to have clean, maintainable code.

  After encountering a warning, ObjectCenter continues reading the file and reports on any additional problems it finds. Files that contain warnings, but no errors, can be used in your programs.

NOTE When you are loading a C++ file, the message text is usually the same as would be produced if you had run the file through **CC**.

Fixing static problems

To fix a problem indicated by a violation message, you first invoke your editor directly on the source code containing the problem. Then after editing the code causing the violation, you build your project to load the changes into ObjectCenter. There are two distinct times when you fix static problems:

- Getting your source code to load successfully

  You deal with load-time errors when you are introducing new source code into your project.

  Because you cannot successfully load a source file that contains a load-time error, you first need to take care of any load-time errors. Since load-time errors are static problems that would also prevent your code from compiling, this stage is strictly analogous to getting your code to compile and link in your standard edit, compile, link cycle.

- Cleaning up source code

  You deal with load-time warnings whenever you do cleanup on your code. If you like keeping your code clean as you work, you can fix load-time warnings at the same time you fix load-time errors. However, you may prefer to deal with cleanup as a final stage of your programming cycle for an individual module or an entire project.

  You can easily postpone dealing with load-time warnings simply by suppressing them in the Error Browser. Later, when you want to deal with them, you can unsuppress them and reload the code.

Using the Error Browser to deal with warnings and errors

When the **Error Browser** button indicates that you have new warning or error messages, you can use the Error Browser to respond in any of the ways in Table 13.

Table 13 How to Respond to Error Browser Warnings and Errors

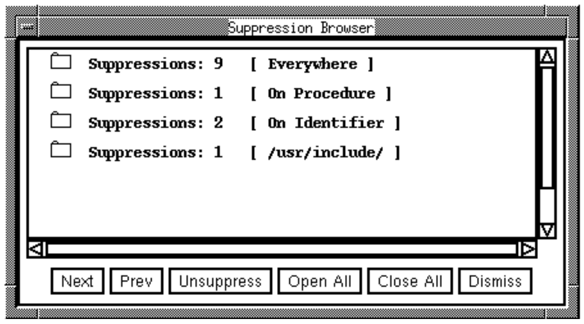
If you want to...	Then take this action...	By doing the following:
Fix the problem immediately.	Examine the warning or error message and invoke your editor on it.	**1** Open the Error Browser, select the Folder symbol for the appropriate file, and examine the violation message. **2** To view the code in the Source area, select the violation message. **3** To invoke your editor on the static problem, select the Edit symbol at the left of the violation message. **4** After fixing the static problems, build your project.
Wait until later to deal with some category of warning messages.	Suppress a category of warning message for a scope you specify. The category of warnings you suppress is no longer reported by ObjectCenter during your session or until you unsuppress it. You cannot suppress errors.	**1** Open the Error Browser, select the Folder symbol for the appropriate file, and select the warning you want to suppress. **2** Display the **Suppress** menu and select the scope you want.

Table 13   How to Respond to Error Browser Warnings and Errors (Continued)

If you want to...	Then take this action...	By doing the following:
Resume dealing with some category of warning message you have suppressed.	Unsuppress the warning for a scope you specify. For warnings that you unsuppress, ObjectCenter again starts reporting that category of warnings.	**1** Open the Error Browser, display the **Suppress** menu, and select **Open Browser**. **2** In the Suppressed Messages window, select the Folder symbol for the appropriate file, and select the warning to unsuppress. **3** Select the **Unsuppress** button.
Not deal at all with a warning or error message during your current session.	Remove a category of warning or error message currently in the Error Browser. The category of violations you remove are taken out of the Error Browser permanently during your session. Unlike suppressing, this does not affect subsequent reporting on this category of violations during your session. For example, messages about violations you removed without fixing will return the next time you build your project. You can remove both warnings and errors.	**1** Open the Error Browser, select the Folder symbol for the appropriate file, and select the violation message you want to remove. **2** Display the **Remove** menu and select the scope you want. You can also choose to remove the selected message or all messages in the window.

Dealing with suppressed messages

The Error Browser has one supporting window, the Suppression Browser, where you deal with warning messages that you have previously suppressed in the Error Browser.



Suppressions: 9 [Everywhere]
Suppressions: 1 [On Procedure]
Suppressions: 2 [On Identifier]
Suppressions: 1 [/usr/include/]

For more information, see the **suppress** entry in the *ObjectCenter Reference*. You can also suppress warnings by customizing your source code; see 'Using lint-style comments to suppress warnings' on page 230.

Understanding scope for suppressing and removing violations

When suppressing warning messages or when removing warning and error messages, you can specify the scope that will apply. Although scope is similar in suppressing and in removing messages, the exact range of scope and how scope applies varies in each case.

For example, suppose you are dealing with the following message:

```
Errors 1 Warnings 6 test11.c (./test11.c)
Line: 11 W#761 The formal parameter ’i’ was not used.
```

Scope for suppressing a warning message

For suppressing, you can specify the following scope:

- Everywhere

  No **Warning #761** is reported again during your session, unless you unsuppress it. This applies to **Warning #761** for any formal parameter, not just **i**.

- In file

  No **Warning #761** is again reported for **test11.c**. This applies to **Warning #761** for any formal parameter, not just **i**.

- At line

  No **Warning #761** is again reported for **line 11** of **test11.c**. This applies to **Warning #761** for any formal parameter on line 11, not just **i**.

- On name

  No **Warning #761** for the formal parameter **i** anywhere in your project is again reported. But **Warning #761** for **j** would be reported.

- In procedure

  All instances of **Warning #761** in the function **do_test** in **test11.c** are suppressed when **do_test** is called.

Scope for removing a warning or error message

For removing messages, you can specify the following scope:

- Everywhere

  All instances of **Warning #761** currently in the Error Browser are removed.

- In file

  All instances of **Warning #761** for **test11.c** currently in the Error Browser are removed.

- At line

  All instances of **Warning #761** for **line 11** of **test11.c** currently in the Error Browser are removed. For example, both of the following messages would be removed:

```
Line: 11 W#761 The formal parameter ’i’ was not used.
Line: 11 W#761 The formal parameter ’j’ was not used.
```

- On name

  All instances of **Warning #761** for the formal parameter **i** currently in the Error Browser are removed. But **Warning #761** for **j** would not be removed.

- In procedure

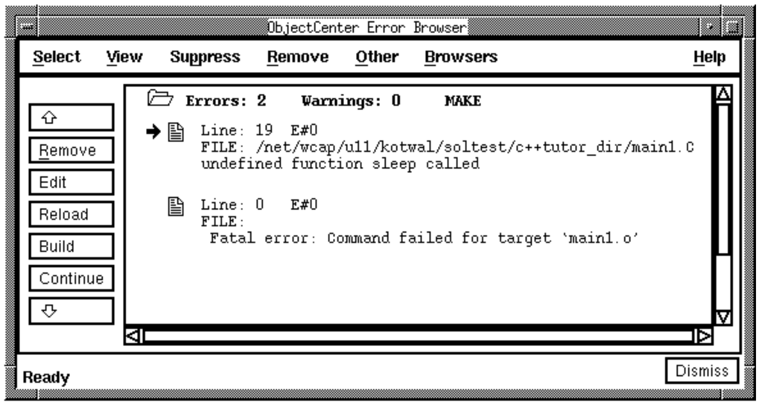
  All instances of **Warning #761** in the function **do_test** in **test11.c** currently in the Error Browser are removed.

Dealing with compiler and make errors

In addition to listing its own load-time warning and error messages, ObjectCenter also captures error messages from your compiler or **make** utility.

When ObjectCenter lists compiler or **make** error messages in the Error Browser, if ObjectCenter can associate them 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.

Here is an illustration of a **make** error message in the Error Browser:


ObjectCenter Error Browser
Select View Suppress Remove Other Browsers Help
Remove
Edit
Reload
Build
Continue
Ready
Dismiss

```
Errors: 2    Warnings: 0     MAKE

Line: 19  E#0
FILE: /net/wcap/u11/kotwal/soltest/c++tutor_dir/main1.C
undefined function sleep called

Line: 0   E#0
FILE:
 Fatal error: Command failed for target 'main1.o'
```

Some compiler and **make** error messages appear only in the Workspace.

For more information, see the **make** entry in the *ObjectCenter Reference*.

Chapter 5 Component debugging

This chapter covers the various aspects of interactively debugging source and object components. It covers the following topics:

- *Using run-time error checking*
- *Using interactive debugging items*
- *Interactive debugging from Workspace break levels*
- *Interactive prototyping and unit testing from the Workspace*

Overview

ObjectCenter's component debugging mode offers a broad range of facilities for interactive debugging. When you execute your code in ObjectCenter, the run-time checker automatically catches dynamic problems in source and instrumented object code. You can use a full set of debugging items for setting breakpoints, actions, and tracepoints. When execution has been stopped by a run-time violation or by a breakpoint, you can work from a Workspace break level. The Workspace also provides direct access to ObjectCenter's interpreter, allowing you to do interactive prototyping and unit testing.

Using run-time error checking

When you execute your code, ObjectCenter automatically checks for dynamic problems. If the run-time error checker finds a dynamic problem, ObjectCenter generates a break level in the Workspace and notifies you with the **Error Browser** button. For run-time errors, you use ObjectCenter's incremental loader/linker and close integration with your editor to correct the source code and quickly load and link the corrections into your ObjectCenter project. For run-time warnings, you can either correct them as you go or suppress them, continue execution, and deal with them at a later time.

Executing your code

Any time you execute your code in component debugging mode, ObjectCenter's run-time checker automatically checks for dynamic problems in source or instrumented object code components. In ObjectCenter you can execute either your entire project or any part of your program at any time. For information on running your project, see 'Running your project' on page 83. For information on executing a part of your code, see 'Interactive prototyping and unit testing in the Workspace' on page 138.

The kinds of dynamic problems ObjectCenter finds

ObjectCenter's run-time error checking automatically detects the following kinds of dynamic problems:

- Losing information during data conversions / assignments
- Calling functions with the wrong number of arguments
- Returning a pointer to an automatic or formal variable
- Trying to free unallocated memory
- Dereferencing a pointer that is out of bounds
- Using a pointer that points to freed memory
- Illegal index into an array
- Division by zero
- Long jump error
- Signals
- C++ casting errors

For a list of all violations that ObjectCenter reports, see the **violations** entry in the online *Reference.*

Dealing with run-time violations

ObjectCenter divides the dynamic problems it finds into two categories—more serious or less serious —and generates a different type of violation message for each kind of problem:

- Run-time error messages

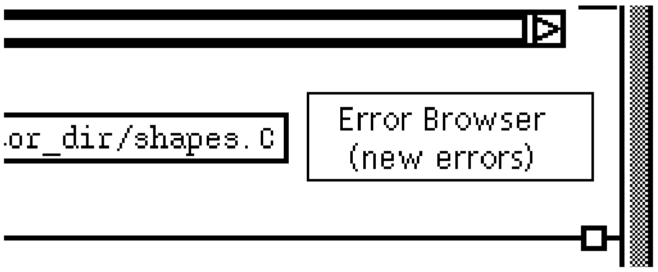
  Run-time error messages indicate serious problems in your executing code. Continuing execution beyond the point where a run-time error occurs would usually result in an invalid program state and might cause a nonrecoverable error to occur. You should take care of run-time errors before continuing your execution.

- Run-time warning messages

  Run-time warning messages indicate dynamic problems that are less serious, and you can usually continue execution beyond them.

Using the Error Browser button

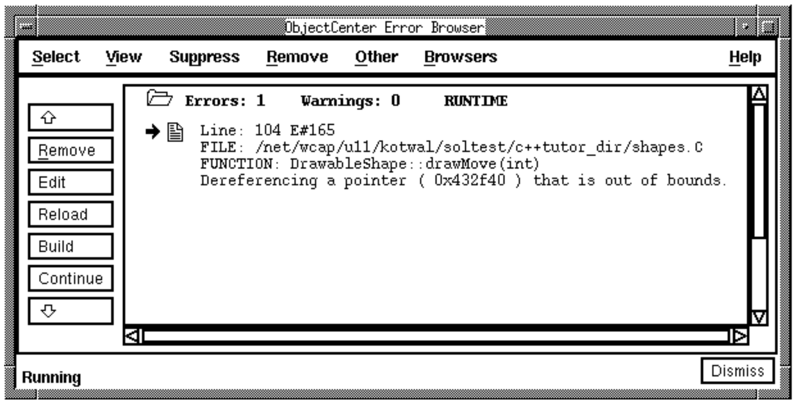
To notify you that new run-time violation messages have arrived in the Error Browser, the Main Window and the Project Browser have an **Error Browser** button that announces new error and warning messages:



To deal with run-time violations, you can open the Error Browser by selecting the **Error Browser** button in the Control Panel of the Main Window or in the Project Browser. You can also select **Error Browser** from the **Browsers** menu in any primary window.

Using the Error Browser

When the **Error Browser** button indicates that you have run-time violations, the Error Browser lists a message summarizing each warning or error. Here is an example.


ObjectCenter Error Browser
Select View Suppress Remove Other Browsers Help
Remove
Edit
Reload
Build
Continue
Running
Dismiss

```
Errors: 1    Warnings: 0    RUNTIME
Line: 104 E#165
FILE: /net/wcap/u11/kotwal/soltest/c++tutor_dir/shapes.C
FUNCTION: DrawableShape::drawMove(int)
Dereferencing a pointer ( 0x432f40 ) that is out of bounds.
```

You can use the Error Browser to respond to run-time violations in the same way as load-time violations. For more information, see the section 'Using the Error Browser to deal with warnings and errors' on page 101.

Continuing past a run-time warning

After examining a new run-time warning, in addition to choosing whether to suppress the warning, you can also choose to continue execution. You can do this by selecting the **Continue** button in the Error Browser.

Fixing dynamic problems

If the violation is a run-time error, you should not continue execution beyond that point until you fix the problem causing the error. You can, however, continue execution from a run-time warning.

CAUTION Continuing past a run-time error can lead to a nonrecoverable error. Although ObjectCenter gives you the option of continuing past an error, be careful about doing this.

When the **Error Browser** button indicates that you have run-time violations, you can deal with them in the following ways:

- To fix a problem immediately, you can examine the violation message in the Error Browser and call your editor directly on the warning or error.
- You can work at the break level using ObjectCenter's code comprehension tools to learn more about the problem. For example, you can examine definitions, look at variable values, examine the calling hierarchy of your program, and display data structures.
- After using your editor to fix an individual violation, you build your project. Reloading a file after a run-time error causes ObjectCenter to reset the Workspace to the top level. You can then run your program again.

TIP: Handling spurious used-before-set messages

Since ObjectCenter has no direct knowledge of the operations executed within object code, run-time violations are difficult to detect when execution moves between source and object code. This is particularly true of dynamic used-before-set violations, since memory can be initialized within object code.

To handle this situation, ObjectCenter stores the value of the **unset_value** option (by default **191**) in each byte of uninitialized data. (This is not true of global and static variables that lack explicit initializers; these variables are initialized to 0, as the C++ language requires at compile time.) The assumption is that if the data is initialized in object code, the data stored will not have the value **191** (octal **277**) stored in any byte. For example:

```
-> char *ptr;
-> ptr = new char;
(char *) 0x38e098 "" /* unset value */
-> *ptr;
(char) ’\277’ /* unset value */
```

This assumption sometimes fails when data is read into memory with an object code library function such as **read()** or **fread()**. This may cause spurious used-before-set warnings if the value stored equals **191**. There are several ways to get rid of these warnings:

- Prevent them by inserting calls to **centerline_untype()**, a built-in ObjectCenter function that marks memory as initialized and valid. The **centerline_untype()** function is similar to the ObjectCenter **touch** command, except that it is easier to use in programs and will not mark unknown memory.

  For an example of using **centerline_untype()**, see the **centerline_untype()** entry in the *ObjectCenter Reference.*
- Suppress the warnings.
- Change the value of the **unset_value** option to **0**. This prevents further dynamic used-before-set warnings. For more information, see 'Using ObjectCenter options' on page 216 and the **options** entry in the *ObjectCenter Reference*.

TIP: Dealing with large numbers of run-time violations

If you are getting an overwhelming number of run-time violations, you can use the following options to temporarily reduce the amount of run-time error checking that ObjectCenter does:

- Set the **save_memory** option to true (set).

  ObjectCenter does not report run-time warnings such as dynamic type mismatches and dynamic used-before-set violations.

- Set the **unset_value** option to **0**.

  ObjectCenter does not report used-before-set violations.

After dealing with your most critical run-time violations, you would restore these options to their original settings and deal with the remaining violations.

For more information, see 'Using ObjectCenter options' on page 216 and information on **save_memory** and **unset_value** in the **options** entry in the *ObjectCenter Reference*.

Using debugging items for interactive debugging

Interactive debugging items

ObjectCenter allows you to interactively set and delete debugging items on code you have loaded. Some debugging items can be set on code in files loaded either in source or object code, some require source code or object code containing debugging information, and some only work in source code.

You can set the following debugging items:

Breakpoint	Stops execution at a line, in a function, or when there is a change of value for an address, variable, or lvalue.
Action	Extends breakpoint functionality by performing customized actions at a line, in a function, or when there is a change in value for an address, variable, or lvalue.
Tracepoint	Turns on line-by-line execution tracing in a specified function or for your entire program.

Setting debugging items in object code

In addition to code loaded in source form, you can also set breakpoints and define actions in code loaded in object form if the code was compiled with the **-g** option (these files contain debugging information). You cannot set breakpoints or actions on an address, lvalue, or variable in object code, and you cannot set tracepoints on or trace through code loaded in object form.

In object code loaded without debugging information, you can set a breakpoint or an action in a function by specifying the function name, but you cannot set a breakpoint or action at a particular line of code.

Setting debugging items on inline functions

You can set debugging items in inline functions in source code.

In object code, you can set debugging items if you compiled the file with **CC**'s **+d** switch. The **+d** switch tells **CC** not to expand inline functions.

Setting breakpoints

You can set as many breakpoints as you like, in as many loaded files as you like.

In source code, you can set a breakpoint on an address, at a location (in a function or at a line), or on an expression (the name of a variable or lvalue). You can use an address for the following items: global variables, allocated data, formal parameters, and automatic variables.

In object code with debugging information, you can set a breakpoint at a location or on an expression. Breakpoints set on an address that your program modifies during execution are unrecognized in ObjectCenter.

In object code without debugging information, you can set a breakpoint only in a function.

Table 14 lists all of the ways to set a breakpoint.

Table 14 Setting a Breakpoint: Methods According to Work Area

Work Area	Ways to Set a Breakpoint
Source area	Place the mouse pointer to the left of a line number and click the Left mouse button to set a breakpoint at that line.
	Place the mouse pointer over a line number at the left of the listed source code and display the **Line Number** pop-up menu. Select **Set Breakpoint Here** to set a breakpoint at that line.
	Place the mouse pointer over listed source code and display the **File Options** pop-up menu. Select **Set Breakpoint**.
Main Window	From the **Debug** menu, select **Set Breakpoint**. In the Set Breakpoint dialog box, specify the scope (at a location, in a function, on an expression or on an address).
Contents window of the Project Browser	Select a function or variable listed in the Contents window and then select the **Stop** button.
Workspace	Use the **stop** command.

You can also use the **centerline_stop()** built-in function to set a breakpoint. See the **built-in functions** entry in the *ObjectCenter Reference*.

For more information on setting a breakpoint, see the **stop** entry in the *ObjectCenter Reference*.

What happens when a breakpoint is triggered

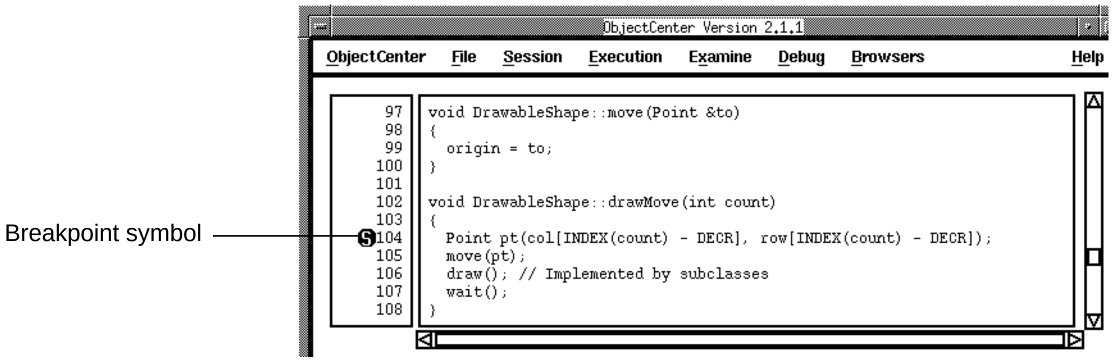
When execution reaches a breakpoint set at a line or in a function, ObjectCenter stops execution, establishes a new break level in the Workspace (see 'Interactive debugging from Workspace break levels' on page 126), and lists the current line of code in the Source area.

When the value at an address with a breakpoint changes, ObjectCenter stops execution, establishes a new break level in the Workspace, and lists the current line of code in the Source area. The same is true when a breakpoint is set on memory within the range you specify with two addresses.

If the value at an address with a breakpoint is modified within compiled code, ObjectCenter does not detect the event, and execution of the program is not interrupted. To avoid spurious messages, the breakpoint does not trigger if the watched address is modified by code you enter in the Workspace.

The Breakpoint symbol in the Source area

When you have set a breakpoint at a line, the Source area displays a Breakpoint symbol to the left of the line number. Here is an illustration:



```
 97  void DrawableShape::move(Point &to)
 98  {
 99    origin = to;
100  }
101
102  void DrawableShape::drawMove(int count)
103  {
104    Point pt(col[INDEX(count) - DECR], row[INDEX(count) - DECR]);
105    move(pt);
106    draw(); // Implemented by subclasses
107    wait();
108  }
```

Setting breakpoints on library functions

You can set a breakpoint in a library function only if the function has been linked in. If you want to set a breakpoint on a library function before running your program, you can use the **link** command to explicitly link unresolved symbols from static libraries. Then set the breakpoint.

Here is an example of setting a breakpoint on a library function without first running the program:

```
-> load bpprog.C
Loading (C++): bpprog.C
-> stop in printf
Cannot set stop or action on an undefined symbol:
’printf’.
-> link
-> stop in printf
stop (1) set at “/lib/libc.sl”, function printf().
->
```

Setting actions

Actions allow you to customize and extend ObjectCenter’s built-in breakpoint functionality by defining some action that will take place when execution triggers it. You set an action by defining a body of C++ or C code or ObjectCenter commands for that action. When execution triggers the action, ObjectCenter evaluates the action body and lists the current line of code in the Source area. Unlike a breakpoint, an action does not automatically stop execution and create a new break level in the Workspace (see 'Interactive debugging from Workspace break levels' on page 126).

You trigger an action in the same ways that you trigger a breakpoint. See 'Setting breakpoints' on page 116.

For more information, see the **action** entry in the *ObjectCenter Reference*.

The same considerations apply to setting actions on library functions as for breakpoints; see 'Setting breakpoints on library functions' on page 118.

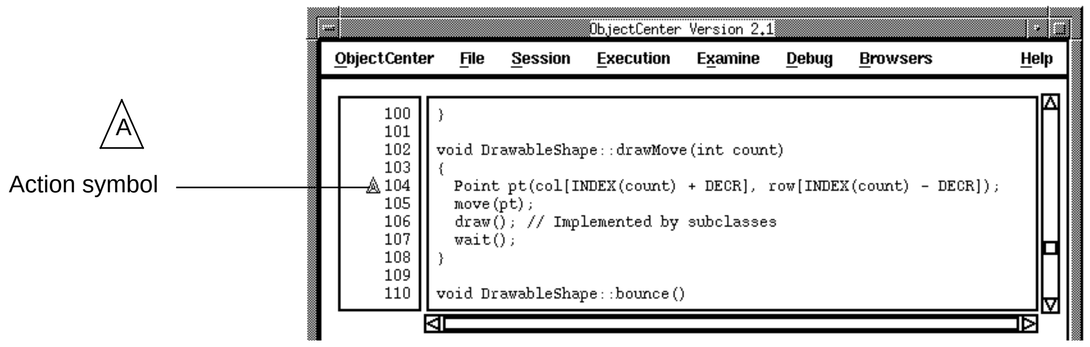
Table 15 lists the ways to set an action.

Table 15 Setting an Action: Methods According to Work Area

Work Area	**Ways to Set an Action**
Source area	Place the mouse pointer over a line number at the left of the listed source code and display the **Line Number** pop-up menu. Select **Set Action Here** to set an action at that line.
	Place the mouse pointer over listed source code and display the **File Options** pop-up menu. Select **Set Action**.
Main Window	From the **Debug** menu, select **Set Action**. In the Set Breakpoint dialog box, specify the scope (at a location, in a function, on an expression, or on an address).
Contents window of the Project Browser	Select a function or variable listed in the Contents window and then select the **Action** button.
Workspace	Use the **action** command.

The Action symbol in the Source area

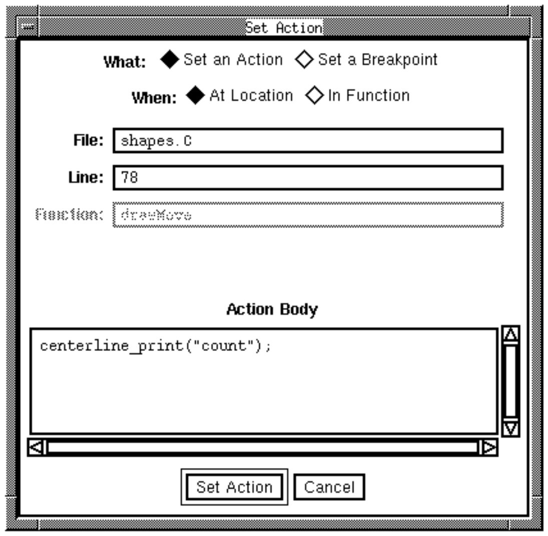
When you have set an action at a line, the Source area displays an Action symbol to the left of the line number. Here is an illustration:



```
100  }
101
102  void DrawableShape::drawMove(int count)
103  {
104    Point pt(col[INDEX(count) + DECR], row[INDEX(count) - DECR]);
105    move(pt);
106    draw(); // Implemented by subclasses
107    wait();
108  }
109
110  void DrawableShape::bounce()
```

Specifying the action body

The body of an action can consist of one or more C++ statements; enclose multiple statements within braces. The following example sets an action on line 78 in **shapes.C**c:


Set Action
What: ◆ Set an Action ◇ Set a Breakpoint
When: ◆ At Location ◇ In Function
File:
Line:
Function:
Action Body
Set Action
Cancel

File: shapes.C

Line: 78

Function: drawMove

```
centerline_print("count");
```

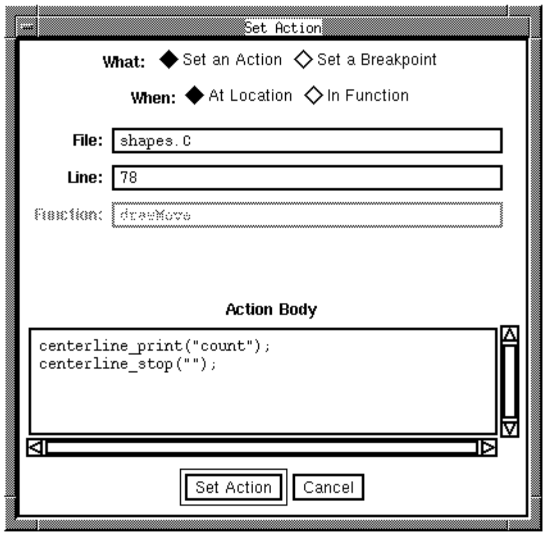
The action causes ObjectCenter to print the value of **count** using a built-in CenterLine function that calls the ObjectCenter **print** command each time line 78 in **shapes.C** is executed.

Alternatively, if you are debugging C code in C mode, actions consist of C statements. You could use the following function in an **action** statement to print the value of **count** in a C program:

```
printf("%d”, count);
```

Unlike breakpoints, actions do not automatically stop execution. To have ObjectCenter stop execution and generate a new break level in the Workspace (see 'Interactive debugging from Workspace break levels' on page 126), you use the CenterLine function **centerline_stop()**, which invokes the ObjectCenter **stop** command.

For example, to have ObjectCenter print the value of **count** and then stop execution, you would use the following definition:



What: [✓ Set an Action] Set a Breakpoint

When: [✓ At Location] In Function

File: shapes.C

Line: 78

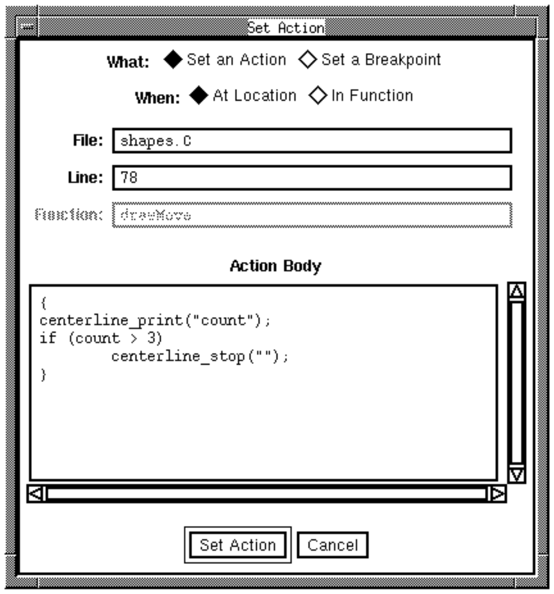
Function: drawMove

```
centerline_print("count");
centerline_stop("");
```

For information on using built-in CenterLine functions, see the **built-in functions** entry in the *ObjectCenter Reference.*

Conditional actions

You can also conditionalize actions. For example, you can have ObjectCenter generate a break level only if certain conditions are true:



What: [✓ Set an Action] Set a Breakpoint

When: [✓ At Location] In Function

File: shapes.C

Line: 78

Function: drawMove

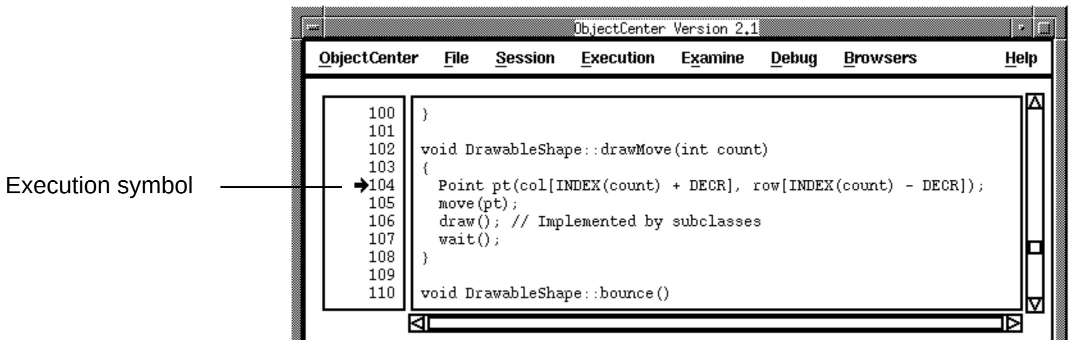
Action Body

```
{
centerline_print("count");
if (count > 3)
        centerline_stop("");
}
```

In addition to printing the value of **count** in the Run window, the action interrupts execution with **centerline_stop()** if **count** is greater than **3**. A function call to **centerline_stop()** in your code is equivalent to using the **stop** command in the Workspace. For more information, see the **built-in functions** entry in the *ObjectCenter Reference*.

Setting tracepoints and tracing program execution

You use tracing to follow the path your program takes as it executes. When tracing is turned on, ObjectCenter continually updates the Source area in the Main Window to show the line of code being executed. The current point of execution is indicated by the Execution symbol, as shown in the following illustration:



```
}

void DrawableShape::drawMove(int count)
{
  Point pt(col[INDEX(count) + DECR], row[INDEX(count) - DECR]);
  move(pt);
  draw(); // Implemented by subclasses
  wait();
}

void DrawableShape::bounce()
```

Table 16 lists all of the ways to set a tracepoint.

Table 16 Setting a Tracepoint: Methods According to Work Area

Work Area	**Ways to Set a Tracepoint**
Source area	Place the mouse pointer over listed source code and display the **File Options** pop-up menu. Select **Set Trace**.
Main Window	From the **Debug** menu, select **Set Tracepoint**. In the Set Tracepoint dialog box, specify the scope (everywhere or in a function).
Workspace	Use the **trace** command.

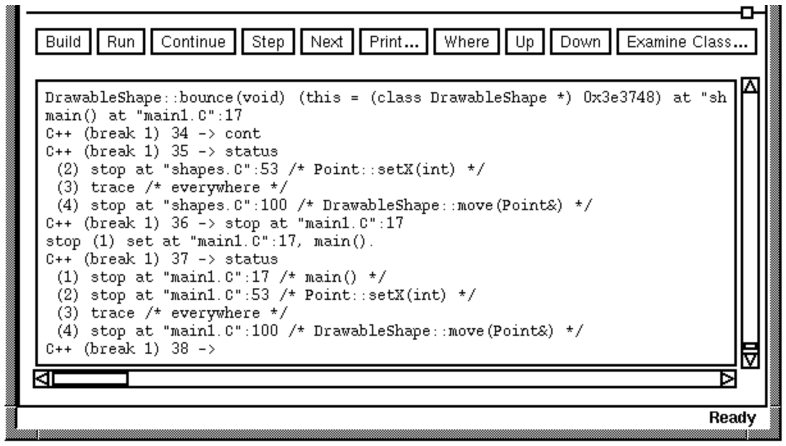
For more information, see the **trace** entry in the *ObjectCenter Reference.*

Examining and deleting debugging items

At any point while working in ObjectCenter, you can see which debugging items are set and delete any of them.

Examining debugging status

To see which debugging items are currently set, display the **Examine** menu in the menu bar of the Main Window and select **Status**. The Workspace echoes the **status** command and lists all the debugging items:



```
DrawableShape::bounce(void) (this = (class DrawableShape *) 0x3e3748) at "sh
main() at "main1.C":17
C++ (break 1) 34 -> cont
C++ (break 1) 35 -> status
 (2) stop at "shapes.C":53 /* Point::setX(int) */
 (3) trace /* everywhere */
 (4) stop at "shapes.C":100 /* DrawableShape::move(Point&) */
C++ (break 1) 36 -> stop at "main1.C":17
stop (1) set at "main1.C":17, main().
C++ (break 1) 37 -> status
 (1) stop at "main1.C":17 /* main() */
 (2) stop at "main1.C":53 /* Point::setX(int) */
 (3) trace /* everywhere */
 (4) stop at "main1.C":100 /* DrawableShape::move(Point&) */
C++ (break 1) 38 ->
```

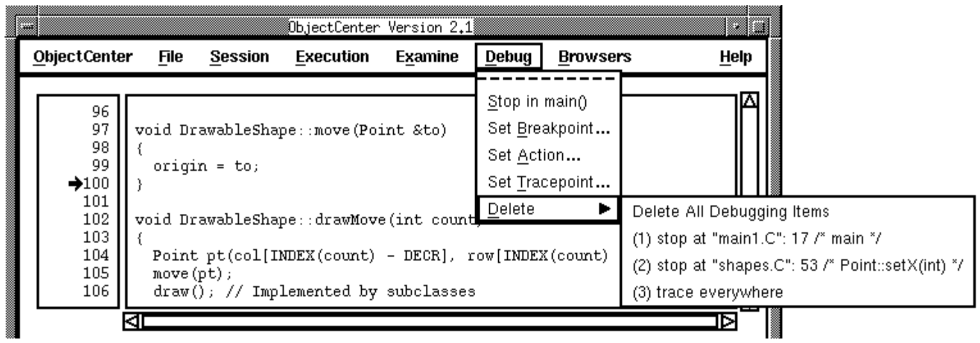
You can also enter the following in the Workspace:

```
-> status
```

For more information, see the **status** entry in the *ObjectCenter Reference*.

Deleting debugging items

To delete debugging items, display the **Debug** menu and select **Delete**. This displays a submenu listing all current debugging items. You can select the item you want to delete, or delete them all at once. See the following illustration for an example.



```

void DrawableShape::move(Point &to)
{
  origin = to;
}

void DrawableShape::drawMove(int count
{
  Point pt(col[INDEX(count) - DECR], row[INDEX(count)
  move(pt);
  draw(); // Implemented by subclasses
```

Table 17 lists all of the ways to delete a debugging item.

Table 17 Deleting a Debugging Item: Methods According to Work Area

Work Area	Ways to Delete a Debugging Item
Source area	Place the mouse pointer over listed source code and display the **File Options** pop-up menu. Select **Delete**. Left click on the Breakpoint or Action symbol itself.
Main Window	From the **Debug** menu, select **Delete**. This displays a menu listing all debugging items currently defined. Select the item you want to delete.
Workspace	Display the **Workspace Options** pop-up menu. Select **Delete**. Use the **delete** command.

For more information, see the **delete** entry in the *ObjectCenter Reference*.

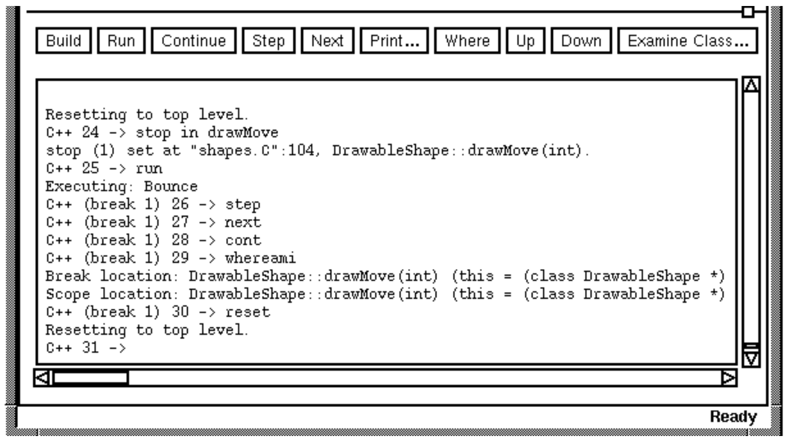
Interactive debugging from Workspace break levels

When you execute a program in ObjectCenter, the Workspace can operate at various levels of nested execution, called **break levels**. When your program begins, the Workspace is at the top level. A break level is generated when execution is interrupted due to one of the following reasons:

- ObjectCenter finds a run-time violation
- You press Control-c
- Execution hits a breakpoint in the program
- A signal occurs that is not being handled by your program

Break levels are an important aspect of ObjectCenter's facilities for source-level debugging. Operating 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.

The current break level is indicated by the Workspace prompt:



```
Resetting to top level.
C++ 24 -> stop in drawMove
stop (1) set at "shapes.C":104, DrawableShape::drawMove(int).
C++ 25 -> run
Executing: Bounce
C++ (break 1) 26 -> step
C++ (break 1) 27 -> next
C++ (break 1) 28 -> cont
C++ (break 1) 29 -> whereami
Break location: DrawableShape::drawMove(int) (this = (class DrawableShape *)
Scope location: DrawableShape::drawMove(int) (this = (class DrawableShape *)
C++ (break 1) 30 -> reset
Resetting to top level.
C++ 31 ->
```

At a break level, you can perform nearly all of the operations supported at the top level of the Workspace. (Some commands such as **build, unload**, and **run** force a reset to the top level.)

Many operations specific to working in a break level are available as commands on the **Execution** menu and as buttons in the Main Window control panel.

Table 18 on page 128 describes the basic Workspace commands that you can use for interactive debugging at a break level.

For detailed information on how to use each of these commands, see the *ObjectCenter Reference* entry for each one. For information on how to create your own buttons for these commands, see 'Creating and managing customized buttons and menu items' on page 220.

Debugging object code

Except for tracing and the **stepout** command, ObjectCenter provides the same debugging capability for object code loaded with debugging information as for code loaded in source form: You can set breakpoints, define debugging actions, and step through the object code.

With object code that is loaded *without* debugging information (either the file was compiled without the **-g** switch or was loaded into ObjectCenter using the **-G** switch), you can stop on or set an action on a function name, but not on a particular line. You cannot step through object code loaded without debugging information.

Debugging inline functions

By default, ObjectCenter's demand-driven code generation is on except for header files. This means that ObjectCenter loads internal information only about code your program uses. Although demand-driven code generation significantly speeds load time, you may want internal information about some of the code your program did not use. For example, if you have defined unused inline functions, you can get internal information about them by loading them without demand-driven code generation (**-dd=off**).

Table 18 Basic Workspace Commands for Interactive Debugging

Workspace Command	Description
action	Specifies statements to execute when execution triggers the action. Allows you to customize conditional breakpoints.
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 variable.
edit	Invokes your editor, positioned at the current line.
expand	Lists the functions that could be called from a C++ statement.
file	Displays and sets the current list location.
info	Displays the name, size, and type of the item associated with an address.
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.
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.

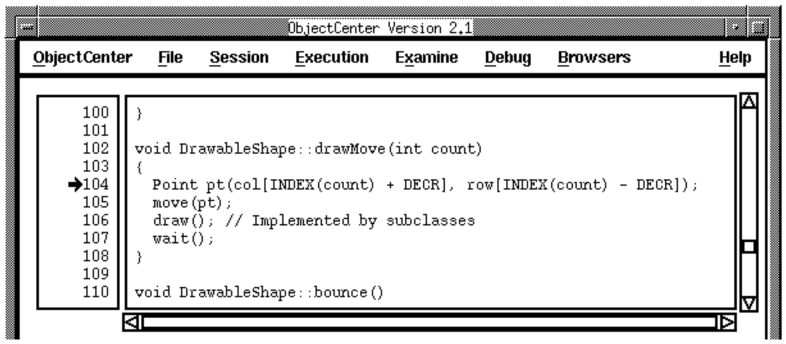
Locations in break levels

While at a break level, ObjectCenter maintains several distinct locations.

The break location

The break location is the location in your program at which execution was stopped and at which execution is resumed when you continue. The values of program variables are determined by their scope at the break location. The break location does not change for a given break level.

The break location is indicated in the Source area by the Execution symbol, as shown in the following illustration:



```
}

void DrawableShape::drawMove(int count)
{
  Point pt(col[INDEX(count) + DECR], row[INDEX(count) - DECR]);
  move(pt);
  draw(); // Implemented by subclasses
  wait();
}

void DrawableShape::bounce()
```

The scope location

The scope location and the break location are identical when the program stops. The scope location, however, can change in response to your actions in the break-level Workspace. For example, you can use the **up** and **down** commands to move around in the execution stack.

The source location

The source location is 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.

Automatic Workspace mode switching

ObjectCenter automatically places you in the Workspace mode that is appropriate for the code you are stopped in. That is, if you are stopped in C++ code, you are placed in C++ mode. If you are stopped in C code, you are placed in C mode. (For more information about ObjectCenter's modes, see 'How the debugging and language modes affect the Workspace' on page 36.)

As you move around in your code (either by moving up or down in the execution stack or by stepping through code as described later), ObjectCenter automatically changes the Workspace mode as appropriate.

Multiple break levels

If execution is interrupted again while you are at a break level, a new break level is generated. Each break level maintains its own break and scope locations.

In the following example, the Workspace contains a call to **err()**, a function defined in the Workspace with an error. When execution continues, a second call to **err()** stops execution at a second break level.

```
C++ 2 -> load_header string.h
Loading (C++): -I. /tmp/OC.4096/string.h
C++ 3 -> void err() {char *d; strcpy (d, “sdfsdf”);};
Warning #769:  d used but not set.
(void)
C++ 4 -> err();

----------------
“workspace”:3, err(void),  (Warning #113)
Using auto variable err(void)`d which has not been
set.
General options:  break/continue/quit/edit/reload

C++ (break 1) 5 -> err();

----------------
“workspace”:3, err(void),  (Warning #113)
Using auto variable err(void)`d which has not been
set.
General options:  break/continue/quit/edit/reload


C++ (break 2) 6 ->
```

The prompt identifies the new break level as number 2. Each break level is numbered according to how many levels deep it is from the top level.

Examining the state of your program

While at a break level, you can examine the state of your program by using ObjectCenter's facilities for code comprehension: the Project Browser, Cross-Reference Browser, Inheritance Browser, and Data Browser (see Chapter 7, 'Visualizing your code'').

Because you only use the Data Browser at run time, it has an especially strong connection with interactive debugging from break levels. Typically, you use the Data Browser when you are at a break level and want to investigate the state of some data structure that is currently in scope. For more information about using the Data Browser, see 'Data Browser' on page 197.

Continuing from a break location

If execution was interrupted because you entered Control-c or because a breakpoint was encountered, you can issue the **cont** command to resume execution from the point at which it had stopped. Selecting the **Continue** button in the Main Window or pressing Control-d are shortcuts for calling **cont** without an argument. In the following example, execution is resumed from break level 2 by pressing Control-d and from break level 1 by calling **cont** without an argument:

```
-> stop in do_bounce
stop (1) set at “bounce.c”:10, do_bounce().
-> stop in store_shape
stop (2) set at “shape.c”:7, store_shape().
-> run
Executing: bounce
(break 1)-> store_shape(500, “triangle”);
(break 2)-> shape;
(char *) 0x195420 “triangle”
(break 2)-> ^D
(void)
                Execution returns to do_bounce() . ..
(break 1)-> shape;
(char *) 0xf7ff8c9c “rectangle”
(break 1)-> cont
                Execution goes into store_shape()...
(break 1)-> shape;
(char *) 0xf7ff8c9c “rectangle”
```

Continuing from a run-time violation

If the break level was generated because of a run-time violation, you can issue the **cont** command with an argument. The argument supplied to **cont** is substituted for the expression containing the error. If no argument is supplied to **cont**, the value of the original expression is used.

In the following example, an attempt was made to dereference and use a null pointer. The value supplied to **cont** is used as the right-hand side of the assignment, and the variable **ptr** is not dereferenced.

```
-> int i, *ptr =0;
-> i = *ptr;
Error #165: Dereferencing a pointer ( ptr ) that is
out of bounds.
 Pointer = 0x0, low bound = 0x0, high bound = 0x0.
(break 1)-> cont 123
(int) 123
-> i;
(int) 123
```

For more information, see the **cont** entry in the *ObjectCenter Reference.*

Resetting from a break level

The **reset** command transfers control from the break level in which it is issued to a higher-level break level. Break levels are specified by numbers.

If no break level is specified, control returns to the top level:

```
(break 3)-> reset
Resetting to top level.
->
```

If **reset** is called with a positive argument, it treats the argument as the number of the break level to which to return. If the argument is negative, the **reset** command backs up the specified number of break levels. For more information, see the **reset** entry in the *ObjectCenter Reference.*

Stepping through your program

You can step through your program from a break level by using the **step** and **next** commands. In the Main Window, these commands are available on the **Execution** menu and as buttons below the Source area.

You can step through code loaded in source form as well as object code that contains debugging information (compiled with the **-g** switch and not loaded using the **load** command's **-G** switch).

As you step through your program, the listing in the Source area is updated to show the current line, as indicated by the Execution symbol. Also, data items in the Data Browser are automatically updated at every break in execution (see 'Data Browser' on page 197).

TIP: Improving performance when stepping through code

If you are stepping through your code and have many data items displayed in the Data Browser, the time spent in updating the Data area 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.

Differences between the step and next commands

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.

On the other hand, the **next** command executes all statements on the current source code line.

The **step** command enters function calls, while **next** does not.

Any breakpoints set within a function are triggered even if you continue over the function call with **next**.

If a numeric argument is passed to **step** or **next**, it is taken as the number of statements or lines to execute before stopping.

The stepout command

In source code, the **stepout** command continues execution until the current function returns. The command is particularly useful if you accidentally entered a function by using **step** instead of **next**.

Displaying the execution stack

The **execution stack** consists of all the functions that are in the process of execution. You can display the execution stack by using the **where** command. The execution stack is displayed starting from the location where execution has stopped in the current break level:

```
(break 2)-> where
stop #1 set in
store_shape(count = (int) 500, shape = (char *)
0x1b2098 “triangle”) at “shape.c”:7
break level #1, line 141
stop #2 set in
main(argc = (int) 1, argv = (char **) 0x1b2058) at
“main.c”:8
```

`(break 2)->` For example, if the user is stopped in **drawMove()** as shown in the screen on on page129, **where** shows the execution stack as:

```
(break 1)-> where
error #24
DrawableShape::drawMove(this = (class DrawableShape
  *) 0x223a98, count = (int) 400) at "shapes.C":90
DrawableShape::doDraw(this = (class DrawableShape *)
  0x223a98) at "shapes.C ":78
DrawableShape::bounce(this = (class DrawableShape *)
  0x223a98) at "shapes.C ":97
main() at "main.C":33
centerline_run((char *) 0x20c8f8 "") builtin function
```

In this example, the user is stopped following an error in the function **drawMove()**, which was called by **doDraw()**, which was called by **bounce()**, which was called by **main()**, which was called by **centerline_run()**, which is the function equivalent of ObjectCenter’s **run** command.

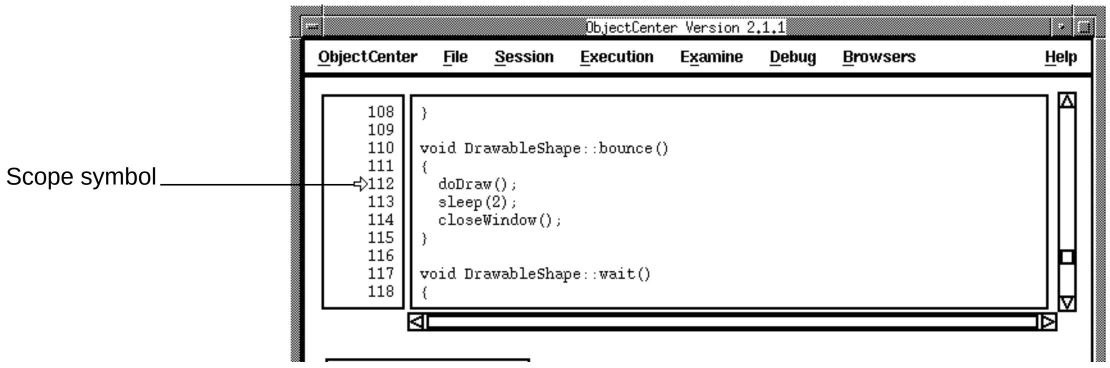
Moving in the execution stack

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 and move to a break level in your source code, ObjectCenter displays the scope location (now different from the break location) in the Source area using the Scope symbol.

As a result of the **whereami** command, the Workspace 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.

If you issue an **up** command after **whereami**, the Source area shows the new scope location. Here's an illustration:



```
}

void DrawableShape::bounce()
{
  doDraw();
  sleep(2);
  closeWindow();
}

void DrawableShape::wait()
{
```

If the user is stopped in the function **drawMove()**, as shown in the previous example, and issues **up**, the scope location is changed to the call to **drawMove()** in **doDraw()**:

```
(break 1)-> up
Scoping to DrawableShape::doDraw() at "shapes.C":78
```

If you get lost

If you have moved around in the execution stack and perhaps have scrolled around in your source code, you might forget where you are. In this situation, issue the **whereami** command.

Displaying static constructors and destructors

There are times when a static constructor or destructor is on the execution stack. For example, you can set a breakpoint on **main()**, then step into a static constructor. The file **static.C** contains this code:

```
// static.C
class foo {
    int i;
public:
    foo () {i = 5;}
};
foo f1;
main () {
}
```

You can step into the static constructor by issuing these commands in the Workspace:

```
-> load static.C
Loading (C++): static.C
-> stop in _main
stop in _main
stop (1) set at "centerline", function _main().
-> run
run
Executing: a.out
Stopped in main() at "static.C":11
    10:
 *  11: main () {
    12: }
(break 1-> step
step
Stopped in static.C::initializer() at "static.C":9
 *   9: foo f1;
(break 1->
```

The top-most function in the execution stack shown above is a static constructor.

ObjectCenter uses the following format when displaying a call to a static constructor:

file::**initializer() at "***file***":***linenum*

Where *file* is the file stopped in, and *linenum* is the location where the static object is declaredSimilarly, ObjectCenter shows static destructors as follows:

file::**deinitializer() at "***file***":***linenum*

Handling signals

Most signals that are generated during execution are caught directly by ObjectCenter. When a signal occurs, execution is stopped and a break level is created.

You can use the **ignore** command to specify which signals should be ignored by ObjectCenter and passed directly to your program. Some signals are ignored by default. Issuing **ignore** without arguments displays the signals currently being ignored:

```
-> ignore
#  Signal     Description
14 SIGALRM    alarm clock
18 SIGCLD     child status change
20 SIGWINCH   window size change
21 SIGURG     urgent socket condition
22 SIGPOLL    pollable event occurred
25 SIGCONT    stopped process has been continued
28 SIGVTALRM  virtual timer expired
29 SIGPROF    profiling timer expired
```

If your program defines a handler for a signal, that signal must be ignored for the handler to receive the signal.

The **catch** command is the converse of the **ignore** command; it specifies which currently ignored signals should be caught by ObjectCenter.

For more information, see the **catch** and **ignore** entries in the *ObjectCenter Reference***.**

Interactive prototyping and unit testing in the Workspace

ObjectCenter's full C++ and C interpreter

In component debugging mode, in addition to handling ObjectCenter commands, the Workspace also functions as a direct interface with the ObjectCenter interpreter. The interpreter operates in either of two language modes: C++ mode or C mode.

Depending on the language mode, you can enter any C++ or C statement at the Workspace to have ObjectCenter execute it immediately. Also, when you execute code from the Workspace, ObjectCenter's run-time error checker automatically checks it for dynamic problems (see 'The kinds of dynamic problems ObjectCenter finds' on page 110).

Interactive prototyping

Because the Workspace allows you to enter any C++ or C statement and immediately execute it, the Workspace supports incremental development through interactive prototyping. You can create code fragments or define variables, functions, and data structures as you go. This means that you have a tool for the highly iterative and explorative phases of your programming style, wherever you choose to bring these phases into your development methodology.

For example, to define a function that adds two integers, you could enter:

```
-> int add (int x, int y)
+> { return x + y; }
-> add (3, 4);
(int) 7
```

Interactive unit testing

Constructing a modular, maintainable application is easier if you can extract a component from its context in the program, test and refine it in isolation, and then merge the modifications back into the program. The Workspace's access to ObjectCenter's interpreter is tailor made for this approach.

Using your editor and ObjectCenter's incremental loader / linker, you can easily create and load a small-to-medium functional unit or a group of related functional units. Using the Workspace, you can test these units individually, as a set, and in interaction with your established project components. Once you have tested a unit, you can then easily integrate it into your project.

For example, with the unit testing capabilities of ObjectCenter's interpreter accessed through the Workspace, you can do the following:

- Load an incomplete program into ObjectCenter. This could be a code fragment you are developing as an enhancement to your application. By linking, you would find the unresolved symbols. Then you could either resolve the symbols by loading other pieces of the application that you need or resolve them artificially by declaring them in the Workspace.

  Using the Workspace, you could also resolve undefined function calls artificially by declaring function stubs. The same applies to resolving data structures by declaring them in the Workspace. You also have the option of ignoring the unresolved references and executing only the section of code that you want to test.

- You can load a complete program into ObjectCenter and invoke individual functions from the Workspace, rather than calling **main()** and running the entire program. Using this approach, you can set a breakpoint in the function that you are executing and stop in that function. Then, while you are stopped in the function, you can test other types of behavior by executing code fragments that simulate the desired behavior. When you are satisfied that the part of your program that you are focusing on supports all the different kinds of behavior you are interested in, you can integrate that code into your project.

Two language modes for the Workspace

Unless you change ObjectCenter's default language (using the **setopt primary_language** option), ObjectCenter's default language is C++. That is, programming statements you enter in the Workspace are considered to be C++ statements and are parsed accordingly. In C++ mode, the interpreter supports the full C++ language, as defined by the *AT&T C++ Language System Product Reference Manual* for Release 3.0 of the AT&T C++ Language System.

The prompt for C++ mode is:

```
C++ 1 ->
```

When in C++ mode, ObjectCenter displays the output of commands using C++ syntax and names.

TIP: Editing Workspace code

During an ObjectCenter session, all C++ definitions you enter are stored in a Workspace scratchpad. To edit Workspace code, use the **edit workspace** command. This command saves your scratchpad to a file named, by default, **workspace.**. To edit the Workspace code, you simply edit this file.

For example, suppose you create a class in the Workspace. You create stubs for other classes and external functions called by the class, and then invoke the methods in the class to test them. After testing your class, you can use the **edit workspace** command to create a file containing the code you defined in the Workspace. ObjectCenter will prompt you to enter a name for the file or accept the default name. For example:

```
-> edit workspace
Appending all workspace definitions to a file.
Default filename is “workspace.” in the current
directory.
Please specify a filename, press Return to accept
default,
or <CTRL-D> to abort:
```

If you want to test a particular set of definitions, edit the file so that it contains the definitions you want to test. Then use the **unload workspace** command to unload all the definitions and objects you created in the Workspace, and use the **source** command to load the definitions in your saved file back into the Workspace. Note that the **source** command will report errors if you’ve unloaded any definitions on which the saved file depends.

If you want to use a new file as source code, add any **#include** lines you need and remove any extraneous lines. For example, some ObjectCenter commands, such as **whatis**, will appear in the file.

When using code developed in the Workspace, remember that static functions and variables, and private and protected member functions, are visible at global scope in the Workspace. As a result, you may have to add friend functions or classes or make static functions externally visible in order to use Workspace sources as a separate file.

You should switch to C mode when you want to:

- Debug C programs by issuing C statements in the Workspace
- See the internals of C++ data structures (such as the virtual function table)

Switching to C mode

To enter C code, the Workspace must be in C mode. In C mode, the interpreter supports the full C language as defined by Kernighan and Ritchie (K&R) and also has support for the ANSI C standard. By default, the interpreter treats C code as K&R C; for information on using ANSI C code, see the **ANSI C** and **config_parser** entries in the *ObjectCenter Reference*.

Use the **cmode** command to switch to C mode:

```
C++ 10 -> cmode
C Workspace Enabled.
C 11 ->
```

Notice that the Workspace prompt is different in C mode:

```
C n ->
```

Specific considerations for entering C code are discussed in the section 'Using the Workspace in C language mode' on page 158.

Switching back to C++ mode

You return to C++ mode by issuing the **cxxmode** command:

```
C 11 -> cxxmode
C++ Workspace Enabled.
C++ 12 ->
```

The language mode does not affect how files are loaded

The language mode you are in does not affect how ObjectCenter loads source files.

For more information about how ObjectCenter loads C and C++ files in C and C++ language modes, see the **language selectio**n entry in the *ObjectCenter Reference*.

Entering C++ and C++ code in the Workspace

When you enter C++ or C code at the Workspace, you terminate each statement or expression with a semicolon. To define a variable and assign a value to it in the Workspace, you could type:

```
C++ -> int i = 5;
C++ -> i;
(int) 5
```

In C mode it would be the same:

```
C -> int i = 5;
C -> i;
(int) 5
```

C++ or C statements input at the Workspace are immediately interpreted and expressions are immediately evaluated. The result is shown in the Workspace below the command line. Definitions are maintained in an internal Workspace scratchpad and are available to be referenced throughout your session.

Using multiple-line statements

C++ or C statements you type in the Workspace can span several lines. To continue the statement on the next line, simply press Return at an appropriate place in the statement. The input prompt changes from **->** to **+>**, indicating that the Workspace is expecting additional input for the statement. For example:

```
-> 123 +
+> 456 +
+> 789;
(int) 1368
->
```

Interpreting the Workspace prompt

Besides the normal Workspace prompt (**->**), there are others for special situations: **+>**, **#>**, and ***>**.

The **+>** prompt appears when the Workspace is waiting for additional code for a multiple-line statement. This often happens when you forget to type a semicolon at the end of a C++ or C statement or expression.

The two other Workspace prompts appear less frequently. The **#>** prompt indicates that the Workspace is waiting for an **#endif** line to end an **#ifdef** statement. The ***>** prompt indicates that the Workspace is waiting for the closing ***/** to end a C comment beginning with **/***.

Responding to errors

ObjectCenter flags run-time warnings and errors you make in the Workspace. For example:

```
-> extern int j;
-> int i;
-> i = j;
Error #155: Undefined variable: ’j’.
(break 1) ->
```

For more information on responding when ObjectCenter displays a run-time violation, see 'Using run-time error checking' on page 109 and 'Interactive debugging from Workspace break levels' on page 126.

Using blocks

Blocks are useful for ensuring that operations performed in the Workspace do not produce adverse side effects or conflict with global variables. All automatic variables declared within the block are local to the block—they cease to exist at the end of the block. You can use these variables for storing values and performing calculations without affecting global variables, even of the same name. For example:

```
-> int i = -1;
-> {
+> int i = 0;
+> while( i <= 10 ) i++;
+> }
(void)
-> i;
(int) -1
```

ObjectCenter does not execute expressions and statements placed within a block until the block is ended with a closing brace (}). Values produced by the expressions and statements within the block are not displayed. Only the **void** value produced at the end of the block is displayed.

Unloading the Workspace

During an ObjectCenter session, all C++ and C definitions you enter from the Workspace are stored in the Workspace scratchpad. You can undefine all C++ and C definitions stored in the Workspace scratchpad by using the **unload workspace** command in the Workspace. For example:

```
-> int i = 99;
-> i;
(int) 99
-> unload workspace
Unloading: workspace
-> i;
Error #742: i undefined.
```

Unloading the Workspace scratchpad does not affect any loaded files or attached libraries. Workspace input history is also unaffected by unloading the Workspace scratchpad.

One reason for unloading the Workspace is to remove a function that was typed in the Workspace. You can also use the command **unload** *function_name* to unload the Workspace scratchpad.

Using the delete operator in the Workspace

The C++ **delete** keyword conflicts with the ObjectCenter **delete** command. To deallocate memory using the **delete** operator in the Workspace, delimit the **delete** statement with parentheses, such as:

```
-> int *iptr = new int[4];
-> (delete iptr);
```

Defining variables and types

You can define and use variables and types directly in the Workspace at any time.

Defining a variable's type and value

To display the type and value of a variable, enter the name of the variable followed by a semicolon.

ObjectCenter evaluates the input and returns its type and value. For statements, the type **void** is displayed. For example:

```
-> int i;
-> i = 16;
(int) 16
-> while (i < 100) i++;
(void)
-> i;
(int) 100
```

For data structures, ObjectCenter displays all nonstatic members:

```
-> struct mystruct {int i; float f; };
-> struct mystruct struct1;
-> struct1;
(struct mystruct) =
{
 int i = 0;
 float f = 0.000000e+00;
}
```

For pointers, ObjectCenter displays the kind of pointer, the address being pointed to, and the data being pointed to:

```
-> char *msg = “hello there”;
-> msg;
(char *) 0x194ea0 “hello there”
```

Declaring types in the Workspace

If an object code file without debugging information is loaded, no information about the types of variables or functions is available.

Consider the file **xyz.C**:

```
int i=4;
int test()
{
return i;
}
```

If this file is loaded into ObjectCenter as an object file compiled with debugging information, you do not need to declare the type of a variable or function that is defined in that file before using it; for example:

```
-> load xyz.o
Loading: xyz.o
-> i;
(int) 4
```

However, if this file is loaded without debugging information (compiled without **-g** or loaded with **-G**), this happens:

```
-> load -G xyz.o
Loading: -G xyz.o
-> i;
Error #742: i undefined.
-> extern int i;
-> i;
(int) 4
```

Defining functions

You can define and use functions directly in the Workspace at any time. When defining a function in the Workspace, you must specify its return type.

For example, to define a function that adds two integers, you could enter:

```
-> int add(int x, int y)
+> {return x + y;}
-> add (3, 4);
(int) 7
```

You can also use a function prototype regardless of whether the **ansi** option is set or unset. For more information, see the **ANSI C** entry in the *ObjectCenter Reference*.

TIP: Understanding unresolved references from a function call in the Workspace

When you call C++ functions in the Workspace, ObjectCenter may notify you of unresolved references unrelated to the functions you call. This is because all static constructors (**sti** routines) are executed, not just the static constructors for the file the function is defined in. Either use the **continue** command or resolve the references.

Defining templates

If you want to define templates in the Workspace, see the “Using templates in the Workspace” section of the **templates** entry in the *ObjectCenter Reference*.

Manipulating class objects in the Workspace

You manipulate class objects in the Workspace basically the same way you manipulate other C++ objects.

Consider this definition of a **String** class in **String.h**:

```
class String{
public:
  String (char*);
  String(int);
  String();
  String(const String&);
  ~String();
  String &operator=(const String&);
  String &operator==(const char*);
  friend String operator+(const String&, const
    String&);
  void getText();
  int getLength();
private:
  int len;
  char *text;
};
```

Assume that **String.C** implements the member functions that are declared in **String.h**. If you load **String.C**, you can interactively manipulate **String** objects in the Workspace. (There are special considerations you need to be aware of if your class definitions are loaded **only** in object form. See 'Workspace troubleshooting' on page 155.)

String.h and **String.C** are provided in the directory that is set up when you go through the ObjectCenter tutorial, so you can try these examples yourself.

Creating class objects

ObjectCenter automatically calls any constructor required to create an instance of a class. It also links in any library modules that are needed.

```
-> load String.C
Loading (C++): String.C
-> String s1="Object";
(class String *) 0x2dcef8 /* (class String) s1 */
-> String s2="Center";
(class String *) 0x2dcf38 /* (class String) s2 */
-> String s3=s1+s2;
(void)
```

Name clashes with ObjectCenter command

If you define a class with the same name as an ObjectCenter command, there is a name conflict when you try to instantiate the class in the Workspace. For example, say you have loaded the following file, **print.C**:

```
class print {
  int i;
  int j;
};
```

When you try to define an instance of the **print** class called **myprint**, you get an error:

```
-> load print.C
Loading (C++): print.C
-> print myprint;
Error #719: myprint undefined.
```

ObjectCenter considered the statement at line 22 to be an invocation of the **print** command—it thought you were asking to print the value of the variable **myprint**. There is no such variable, so an error resulted.

To define an instance of such a class, simply use the **class** keyword at the beginning of the declaration (normally the keyword is optional with the C++ language). For example:

```
-> class print myprint;
(class print *) 0x223ec0 /* (class print) myprint */
-> myprint;
(class print) =
{
 int i = 0;
 int j = 0;
}
-> print myprint
(class print) =
{
 int i = 0;
 int j = 0;
}
```

Accessing class objects

You display class objects the same way you display other data structures in ObjectCenter; that is, by specifying the object's name followed by a semicolon. ObjectCenter displays all nonstatic data members for a given class object.

```
-> String s3;
Linking from '/usr/lib/libC.a' .....Linking completed
Linking from '/lib/milli.a' ... Linking completed.
(class String *) 0x40124b98 /* (class String) s3 */
-> s3;
(class String) =
{
 int len = 12;
 char *text = 0x1fbcc0 "ObjectCenter";
}
```

You access class members using the same syntax used in C++ programs:

```
-> s3.getLength();
(int) 12
-> s3.len;
(int) 12
-> s3.text;
(char *) 0x1fbcc0 "ObjectCenter"
-> String *sptr = new String;
(class String *) 0x1fbc80 /* (class String)
(allocated) */
-> sptr->text="A string";
(char *) 0x1fbca0 "A string"
-> *sptr;
(class String) =
{
 int len = 0;
 char *text = 0x1fbca0 "A string";
}
```

NOTE There is no access protection in the Workspace. You can reference any member function or data member of any class, even those defined as protected or private.

Deleting class objects

The C++ **delete** keyword conflicts with the ObjectCenter **delete** command. To deallocate memory using the **delete** operator in the Workspace, delimit the **delete** statement with parentheses, such as:

```
-> String *sptr = new String("ObjectCenter");
(class String *) 0x2c5920 /* (class String)
(allocated) */
-> (delete sptr);
(void)
```

How the Workspace displays class objects

When ObjectCenter displays a class object it displays all nonstatic data members in that object. For example:

```
-> String s1="Object";
(class String *) 0x2dcef8 /* (class String) s1 */
-> String s2="Center";
(class String *) 0x2dcf38 /* (class String) s2 */
-> String s3=s1+s2;
(void)
-> s3;
(class String) =
{
 int len = 12;
 char *text = 0x1fbcc0 "ObjectCenter";
}
```

Displaying pointers to class objects

When displaying pointers to class objects, ObjectCenter displays the type of pointer, the address being pointed to, the type of the object currently being pointed to, and the object being pointed to:

```
-> String *sptr;
-> sptr =
(class String *) 0x2e3148 /* (class String) s3 */
```

In the above case, **sptr** is defined as a pointer to **String** and is currently pointing to a **String** (**s3**). 0x2e3148 is the address being pointed to. Consider the following case, where a pointer to **Parent** (**pptr**) is defined, then pointed to an instance of class **Child**, which is derived from **Parent**:

```
class Parent {
 int i;
};

class Child : public Parent {
 int j;
};
-> Parent p1;
-> Parent *pptr = &p1;
-> pptr;
(class Parent *) 0x2e2b60 /* (class Parent) p1 */
-> Child c1;
C++ 63 -> pptr = &c1;
(class Parent *) 0x308e88 /* (class Child) c1.Parent::i */
```

When you dereference a class pointer, ObjectCenter displays the full object being pointed to:

```
-> *pptr;
(class Child) =
{
 int Parent::i = 0;
 int j = 0;
}
```

Here, ObjectCenter displays the **Child** object being pointed to. If you don't want to display the run-time type of a pointer, but rather display the compile-time (or definition-time) type of the pointer, unset the option **print_runtime_type**, which is set by default:

```
-> unsetopt print_runtime_type
-> *pptr;
(class Parent) =
{
 int i = 0;
}
```

Displaying pointers to data members

For pointers to class data members, ObjectCenter displays the kind of pointer, the address pointed to, the type of object pointed to, and the data member pointed to:

```
-> String s5="A string";
(class String *) 0x2066c8 /* (class String) s5 */
-> int *ptr;
-> ptr = &s5.len;
(int *) 0x2066c8 /* (class String) s5.String::len */
-> ptr;
(int *) 0x2066c8 /* (class String) s5.String::len */
-> *ptr;
(int) 8
```

Displaying the inheritance hierarchy

ObjectCenter shows inheritance when displaying members in a derived class.

Consider the following class hierarchy, where class **Two** is derived from class **One**, class **Three** is derived from class **Two**, and so on:

```
class One {protected:  int a;};
class Two : public One {protected:  int b;};
class Three : public Two {protected:  int c;};
class Four : public Three {protected:  int d;};
class Five : public Four {protected:  int e;}
```

ObjectCenter displays an instance of the **Five** class as follows:

```
-> load hierarchy.C
Loading (C++): hierarchy.C
-> Five myfive;
-> myfive;
(class Five) =
{
 int Four:Three:Two:One::a = 0;
 int Four:Three:Two::b = 0;
 int Four:Three::c = 0;
 int Four::d = 0;
 int e = 0;
}
```

If a member is defined in the class itself, ObjectCenter simply shows the member. For example, the member **e** is defined locally by the **Five** class, so ObjectCenter shows:

```
int e ...
```

If a member is inherited, ObjectCenter displays the class in which the member is defined using the syntax **defining_class::member.**

For example, the member **d** is defined in **Four**, so ObjectCenter shows:

```
int Four::d ...
```

If an inherited member is defined by a higher-level class than the direct base class, ObjectCenter displays the entire path of inheritance as **class1:class2:class3: ... defining_class::member.**

For example, the member **c** is defined in the class **Three**. The **Five** class inherits that member through its base class **Four**. So ObjectCenter shows the inheritance path as:

```
int Four:Three::c ...
```

Similarly, the member **b** is inherited through the path from class **Four** through class **Two**, so ObjectCenter shows:

```
int Four:Three:Two::b ...
```

NOTE ObjectCenter uses single colons (:) to indicate the inheritance path and uses the scoping operator (::) to indicate the class that actually defines a member.

The inheritance path is for display purposes only

While ObjectCenter shows the inheritance path when displaying class members, you don't specify the path when identifying class members in the Workspace.

For example, the following is an error:

```
-> myfive.Four:Three::c;        // ERROR
```

You identify class members the same way C++ does, by simply specifying the class name, the selection operator (.), and the member name, such as:

```
-> myfive.c;
(int) 0
```

Though note that the following **is** valid C++ syntax:

```
-> myfive.Three::c;
(int) 0
```

In some, but not all, cases, that syntax can resolve an ambiguity that might exist when accessing a data member.

Suppressing the display of the inheritance path

You can tell ObjectCenter not to display the complete path of inheritance for inherited members by unsetting the option **show_inheritance**, which is set by default.

In the following example, **show_inheritance** is unset, so only the defining class is shown for members.

```
-> unsetopt show_inheritance
-> myfive;
(class Five) =
{
 int One::a = 0;
 int Two::b = 0;
 int Three::c = 0;
 int Four::d = 0;
 int e = 0;
}
```

Notice that ObjectCenter is no longer showing *how* **myfive** inherited members; it is simply showing the class that defines each member.

Suppressing the display of inherited members

By default, ObjectCenter displays all nonstatic data members of an object. Using ObjectCenter options, you can change whether ObjectCenter displays inherited members and static members.

If you unset the option **print_inherited** (which is set by default), ObjectCenter displays only locally defined members of objects:

```
-> myfive;
(class Five) =
{
 int One::a = 0;
 int Two::b = 0;
 int Three::c = 0;
 int Four::d = 0;
 int e = 0;
}
-> unsetopt print_inherited
-> myfive;
(class Five) =
{
 int e = 0;
}
```

ObjectCenter now only displays **e**, which is the only data member that is defined in class **Five**.

Turning on the display of static members

Similarly, by setting the option **print_static** (unset by default), you can display static members.

Given this class:

```
class test {
  static int count;
  int i;
};
```

ObjectCenter displays this:

```
-> test test1;
-> test1;
(class test) =
{
 int i = 0;
}
-> setopt print_static
-> test1;
(class test) =
{
 static int count = 0;
 int i = 0;
}
```

Workspace troubleshooting

Because of the complexity of C++ (particularly the use of constructors, inline functions, and virtual functions), there are several situations when working in the Workspace can get confusing.

In particular, you might be told that you have unresolved references in your program, and it is not immediately obvious what the problem is. This section describes some of these situations.

Unresolved references with static constructors

If you call a function in the Workspace, *all* static constructors (**sti** routines) are executed, not just the static constructors for the file the function is defined in. (This is because it is not possible to determine what can be called when a statement is executed; therefore, ObjectCenter needs to assume that everything might get called.)

This can be confusing. For example, you load several C++ files and call a function. ObjectCenter might report unresolved references from **sti** routines that are completely unrelated to the function you are calling.

Unresolved references with inline functions

It is possible to run a program without encountering any unresolved references, but then discover unresolved references when you use the Workspace to call functions in the program. This occurs if your program never used the functions you are trying to call in the Workspace.

By default, ObjectCenter uses demand-driven code generation (**-dd=on** with the **load** command). If demand-driven code generation is on, ObjectCenter loads only the code your program uses. Setting **-dd=off** eliminates unresolved references when you use the Workspace to call functions your program never used.

For example, you can have inline functions your program never calls. When you run your program, these inline functions are not called, so you do not discover any undefined functions that are called by those inline functions. However, when you call such an inline function in the Workspace, ObjectCenter discovers any undefined functions called by the inline function, generates a break level, and reports unresolved references.

You must set **-dd=off** to generate information about these functions and call them in the Workspace.

Using virtual functions with incomplete programs

When working with incomplete programs, virtual functions can reduce the usefulness of the Workspace. One problem results from the initialization of the **vptr** inside a constructor. The pointer is initialized to the **vtbl** for the class, which references **all** virtual functions defined for the class.

Consider this class definition in **virtual.h**:

```
class p {
  virtual one ();
  virtual two ();
};
```

and this file, **virtual1.C**, which defines **one()** but not **two()**:

```
#include "virtual.h"
int p::one() {
  return 1;
};
```

If you load **virtual1.C** and try to create an instance of the class, you get a warning:

```
-> load -dd=off virtual1.C
Loading (C++): virtual1.C
-> p p1;
Error #156: Calling undefined function p::p(void).
(break 1)-> unres
Undefined symbols:
int p::two()
(break 1)-> cont
Error #156: Calling undefined function p::p(void).
(break 1)-> p1;
(class p) =
{

(break 1)-> cmode
C Workspace Enabled.
C (break 1)-> p1;
(struct p) =
{
 struct __mptr *__vptr__1p = 0x0;
}
C (break 1)-> cxxmode
C++ Workspace Enabled.
C++ (break 1)->
```

Here we have created an instance of the class **p**, which contains virtual functions. The unresolved warning results from not all the virtual function definitions being loaded into ObjectCenter. In this case, you can ignore the warning until **p::two()** is referenced in a call or pointer to a member assignment.

Optimization in the C++ language

A more serious case arises from the optimization included in C++ beginning with Release 2.0 of the AT&T C++ Language System. (This optimization is described in the paper "Virtual table optimizations in C++ 2.0," by Andrew Koenig and Stan Lippman in the March 1990 issue of The *C++ Report*.)

The optimization attempts to create one definition of a class's **vtbl** per executable. Before Release 2.0, this could only be accomplished by using the **CC** switches **+e0** and **+e1**.

The algorithm is to generate a **vtbl** definition for a class if the module being compiled contains the definition of the first non-inline virtual member function declared in the class.

For example, now load this new file, **virtual2.C**, which defines **two()**:

```
#include "virtual.h"
int p::two() {
  return 1;
}
```

we find the **vtbl** for **p** defined in only one file:

```
C++ 9 -> load -dd=off virtual2.C
Loading (C++): virtual2.C
C++ 10 -> whereis __vtbl__1p
"virtual1.C" struct __mptr __vtbl__1p[4], initialized
```

This presents a problem in the Workspace. If you attempt to instantiate a class that contains virtual functions, and the file that contains the **vtbl** is not loaded, the instance will not be initialized properly:

```
C++ 11 -> unload user
Unloading: virtual1.C
Unloading: virtual2.C
C++ 12 -> unload Workspace
C++ 13 -> load -dd=off virtual2.C
Loading (C++): virtual2.C
C++ 14 -> p p1;
Warning #73: Calling function p::p() which contains
unresolved references. Use the 'unres' command to
list all undefined symbols.
C++ (break 1) 15 -> unres
Undefined symbols:
   extern struct __mptr vtbl__1p[];
C++ (break 1) 16 -> cont
(class p *) 0x399398 /* p1 */
```

```
C++ 17 -> cmode
C Workspace Enabled.
C 6 -> p1;
(struct p) =
{
 struct __mptr *__vptr__1p = 0x0;
}
```

You need to understand the optimization in the C++ Language System and load the appropriate files to create a usable Workspace.

Using the Workspace in C language mode

You can load, run, and debug C programs as well as C++ programs in ObjectCenter. In order to enter standard C statements in the Workspace, you use ObjectCenter's *C mode***.**

Viewing C definitions

With files loaded as C source code, no C definitions or external definitions are visible in the Workspace in C++ mode. Consider the C file **c_int.c**, which contains only this definition:

```
int c_var;
```

Look at what happens if you load **c_int.c** and ask for the value of the defined integer:

```
C++ 55 -> load -C c_int.c
Loading (C): c_int.c
C++ 56 -> c_var;
Error #719: c_var undefined.
```

The workaround is to use C mode:

```
C++ 57 -> cmode
C Workspace Enabled.
C 58 -> c_var;
(int) 0
```

Differences between C++ mode and C mode

When you are in C mode, statements you enter in the Workspace are considered to be standard C statements and are parsed accordingly.

Also, the output of Workspace commands is shown using C, not C++, syntax. That means that all C++ identifiers are shown in *mangled form*, after they have been translated from C++ code to C code, prior to compilation.

Take a look at what happens when a class instance is created in C++ mode, then the user switches to C mode:

```
C++ 4 -> load String.C
Loading (C++): String.C
C++ 5 -> String s1="This is a string";
(class String *) 0x1fc280 /* s1 */
C++ 6 -> s1;
(class String) =
{
 int len = 16;
 char *text = 0x21f190 "This is a string";
}
C++ 7 -> cmode
C Workspace Enabled.
C 8 -> s1;
(struct String) =
{
 int len__6String = 16;
 char *text__6String = 0x21f190 "This is a string";
}
```

As shown on line 8, the member names are shown in mangled form: **len** becomes **len__6String** and **text** becomes **text__6String**.

Attempts to access the members using the C++ names fail when in C mode:

```
C 10 -> s1.len;
Error #713: 'len' is not a member of type (struct
String).
C 11 -> cxxmode
C++ Workspace Enabled.
C++ 12 -> s1.len;
(int) 16
```

When in C mode, you access members by using their mangled names:

```
C 13 -> s1.len__6String;
(int) 16
```

Also, using a C++ construct while in C mode fails. In the following example, an attempt was made in C mode to concatenate two strings using the overloaded + operator.

```
C 32 -> s1+s2;
Error #593: Illegal argument type:
     (struct String) + (struct String)
```

There are several other differences between working in C++ mode and C mode.

Displaying references

In C++ mode, when you display a C++ reference type, ObjectCenter returns the value of the reference type:

```
C++ 70 -> int i = 5;
C++ 71 -> int &ref = i;
C++ 72 -> ref;
(int) 5
```

In C mode, ObjectCenter shows you what the reference type is an alias for. In this case, **ref** is an alias for **i**.

```
C++ 73 -> cmode
C Workspace Enabled.
C 74 -> ref;
(int *) 0x206050 /* i */
```

Displaying static objects

ObjectCenter displays static objects when in C++ mode (if the **print_static** option is set), but not in C mode. (This is because static objects don’t actually exist internally in the C structure.)

Given this class:

```
class test {
  static int count;
  int i;
};
```

ObjectCenter displays this:

```
C++ 70 -> class test { static int count; int i; };
C++ 71 -> int test::count = 0;
C++ 72 -> setopt print_static
C++ 73 -> test test1;
C++ 74 -> test1;
(class test) =
{
 static int count = 0;
 int i = 0;
}
C++ 75 -> cmode
C Workspace Enabled.
C 76 -> test1;
(struct test) =
{
 int i__4test = 0;
```

Displaying virtual table pointers

In C++ mode (the default), ObjectCenter doesn't display virtual table pointers; in C mode it does. For example, given this class definition:

```
class foo {
  virtual void func();
  int i;
};
```

ObjectCenter displays this:

```
C++ (break 1) 44 -> foo f2;
(class foo *) 0x308ec8 /* (class foo) f2 */
C++ (break 1) 45 -> f2;
(class foo) =
{
 int i = 0;
}
C++ (break 1) 46 -> cmode
C Workspace Enabled.
C (break 1) 47 -> f2;
(struct foo) =
{
 int i__3foo = 0;
 struct __mptr *__vptr__3foo = 0x309e90 /* (struct
__mptr) __vtbl__3foo[0] */;
}
```

The virtual table pointer is identified at the end as **struct __mptr *__vptr__3foo.**

Displaying virtual base class pointers

Pointers for virtual base classes are shown in C mode, but not in C++ mode.

Displaying members in classes with multiple inheritance

Inherited members created using multiple inheritance are shown as members in C++ mode and as sub-objects in C mode:

```
C++ 51 -> class a {int i;};
C++ 52 -> class b {int j;};
C++ 53 -> class c : public a, public b {int k;};
C++ 54 -> c c1;
C++ 55 -> c1;
(class c) =
{
 int a::i = 0;
 int b::j = 0;
 int k = 0;
}
```

```
C++ 56 -> cmode
C Workspace Enabled.
C 57 -> c1;
(struct c) =
{
 int i__1a = 0;
 struct b Ob =
 {
 int j__1b = 0;
 };
 int k__1c = 0;
}
```

Chapter 6 Process debugging

This chapter describes ways for you to get information about your program's structure and current state. It covers the following topics:

- *How you work in process debugging mode*
- *Getting into process debugging mode and specifying a target*
- *Differences between the two debugging modes*

Overview

You use ObjectCenter's process debugging mode for interactive debugging of an externally linked executable as an autonomous process.

For the most part, in process debugging mode, you conduct interactive debugging and data-level code comprehension in the same way you do in component debugging mode. This chapter concentrates on those aspects of interactive debugging that are unique to process debugging mode and explains the differences between the two debugging modes.

How you work in process debugging mode

Process debugging mode allows you to debug externally linked executables. This provides you with a superior programming environment for traditional source-level debugging. You get the interactive debugging features of source-level debuggers such as **gdb**, plus the unique code comprehension facilities offered by ObjectCenter's Data Browser, Source area, and Workspace commands. Because you use the same command set and access functionality through the same GUI, both process and component debugging modes combine to offer you a unified programming environment. At any time, you can switch between the two debugging modes to suit your current programming objectives.

When you use process debugging mode

There are times you need to use process debugging mode to do things not possible in component debugging mode, such as tracking down a bug that depends on the way your linker lays out storage or a bug that depends on multiprocess interactions.

Typically, however, you use process debugging mode when you simply want to get a quick check of your program's behavior and are willing to sacrifice the extra benefits of using component debugging mode. Using an externally linked executable, you get faster setup time for your debugging session and faster execution speed for your program running in ObjectCenter.

But this increase in setup and execution speed for process debugging mode comes at the expense of some of ObjectCenter's most powerful features that are only available in component debugging mode: automatic error checking, graphical views of the calling hierarchy using the Cross-Reference Browser, graphical views of the inheritance hierarchy using the Inheritance Browser, access to all categories of definitions contained in a file using the Contents window, and access to the interactive C++ interpreter or C interpreter.

Ways to target an externally linked executable

In process debugging mode, you can debug an externally linked executable in any of three ways: as an executable alone, or with a corefile or running process. You choose among these ways according to your current programming objective.

Using an executable alone

If you are simply interested in going after a known bug in the simplest method possible, you start a process debugging mode session and target an executable alone. When you target an executable alone, ObjectCenter starts the target executable as a new process and pauses execution at the first instruction of the program.

Using an executable with a corefile

If your executable has crashed and dumped core, you can target the corefile and go immediately to the point of the crash.

When you use an executable with a corefile, ObjectCenter recreates the program state, halted at the point where your program faulted. Since the corefile is an image of your application's memory at the time of the crash, you have access to the complete program state when the crash occurred. However, you cannot restart or continue execution from a corefile.

Using an executable with running process

If you are up against a bug that occurs in an event-driven or multiprocess application, you can use an executable with a running process.

When you use an executable with a running process, ObjectCenter attaches to the running process and process execution is suspended. (If you prefer, you can first suspend the running process from the shell using the UNIX **kill -STOP** command and attach to the process later.) This allows you to explore the program in the context of complex or transient states that would be difficult to recreate any other way. You can then use the interactive debugging facilities to change the state, resume execution, and debug the program.

Getting into process debugging mode and specifying a target

Selecting process debugging mode at startup

If you are not already in the ObjectCenter environment, you can start ObjectCenter in process debugging mode by using the **-pdm** switch at the shell command line:

```
   % objectcenter -pdm
```

When you start up in process debugging mode, ObjectCenter reads the global startup file **ocenterinit** and the local startup file **.pdminit** (rather than the local startup file **.ocenterinit**). For information about customizing these startup files, see 'Using ObjectCenter startup files' on page 213.

Running process debugging mode only means being without access to **clms** (CLIPC Message Server), the process that manages the interprocess communication among all Ascii ObjectCenter application services. With process debugging mode only, you have no access to the GUI as a way of switching to component debugging mode.

Once you are in an ObjectCenter session in one debugging mode, you can easily switch to the other mode. However, switching modes means that you begin a new session; the state of your debugging session is not carried over from one mode to the other.

To switch from one mode to another, display the ObjectCenter menu in the Main Window and select **Restart Session**. This opens the Restart Environment dialog box. You can specify the items listed in Table 19 on page 168.

When you begin process debugging mode the first time, the Workspace displays a startup message that you can have ObjectCenter suppress for subsequent sessions. The Workspace prompt indicates that ObjectCenter is in process debugging mode (pdm):

```
   pdm ->
```

Table 19   Switching Debugging Modes

To do this...	Take this action...
Switch to process debugging mode	Select **Process Debugger** for the Run-time Engine.
Switch to component debugging mode	Select **Component Debugger** for the Run-time Engine.
Use a startup directory that is different from ObjectCenter's current working directory	Specify the pathname.
Specify a target for process debugging mode or a file to load (such as a project file or source or object components) in component debugging mode	Specify these as arguments.
Initialize the new session using the global initialization file **ocenterinit**	Select **Load Global Initialization File.** For more information, see 'Using ObjectCenter startup files' on page 213.
Initialize the new session using the local cdm initialization file **.ocenterinit** or the local pdm initialization file **.pdminit**	Select **Load Local Initialization File**. For more information, see 'Using ObjectCenter startup files' on page 213.

Specifying a debugging target

You specify your debugging target by using the ObjectCenter **debug** command from the Workspace.

Specifying an executable alone

You use an executable as your debugging target by giving the name of the executable as an argument. For example:

```
pdm -> debug my.a.out
```

Using an executable with a corefile

You use an executable with a corefile by giving the name of an executable and its associated corefile as arguments. For example:

```
pdm -> debug my.a.out my_core
```

Using an executable with a running process

You use an executable with a running process by giving the name of an executable and the process id number for the associated running process. For example:

```
pdm -> debug my.a.out 49172
```

For more information about specifying a debugging target, see the **debug** entry in the *ObjectCenter Reference*.

Differences between the two debugging modes

Wherever possible, ObjectCenter maintains consistency of use in both component debugging mode and process debugging mode. Except for the differences discussed here, you can approach interactive debugging in process debugging mode in the same way described for debugging in component mode.

Elements of the GUI not supported in process mode

When in process debugging mode, those elements of the GUI, such as menu items and buttons, that do not apply to this mode are grayed out. The most prominent elements of the GUI not supported in process debugging mode are the Project, Cross-Reference, Inheritance, and Options Browsers.

ObjectCenter commands

Most ObjectCenter commands are the same for both debugging modes. However, there are some differences. Some commands apply only to component debugging mode, other commands apply only to process debugging mode, and some commands use slightly different syntax or have slightly different behavior depending on which mode you invoke them in.

For a list of the ObjectCenter commands supported in process debugging mode, along with a description of any differences between the way each command works in process debugging mode compared to component debugging mode, see the **pdm** entry in the *ObjectCenter Reference*.

You can also enter the following in the Workspace:

```
pdm -> help
```

ObjectCenter options

Most ObjectCenter options do not apply to process debugging mode, and the Options Browser is not available. There are two options available in process debugging mode: **class_as_struct** and **full_symbols**. Both are described in the options entry in the *ObjectCenter Reference*.

Error checking

In process debugging mode, ObjectCenter does not perform automatic load-time or run-time error checking.

Interactive debugging

Most of the interactive debugging facilities operate transparently between component and process debugging modes.

To set actions in process debugging mode, you use the **when** command from the Workspace, rather than the **action** command. For more information, see the **when** entry in the *ObjectCenter Reference*. For more information on using the other debugging items, see 'Using debugging items for interactive debugging' on page 115.

Working at break levels

In process debugging mode, you work at break levels in much the same way as you do in component debugging mode (see 'Interactive debugging from Workspace break levels' on page 126).

However, since the Run-time Engine in process debugging mode does not include the C++ interpreter or C interpreter, from break level 1 you cannot invoke a new line of execution and generate a nested break level. For interactive debugging in process debugging mode, the Workspace is either at the top level or at break level 1.

Code comprehension

Process debugging mode offers the same code comprehension features in the Data Browser, Source area, and Workspace commands (see 'Data Browser' on page 197 and 'Pop-up menu and Workspace commands' on page 206).

When you display the Data Browser in process debugging mode, you cannot use it to change the values in the elements of data structures as you can in component debugging mode. Since the Project Browser, Contents window, and Cross-Reference Browser only apply to code comprehension for project components, you do not use them in process debugging mode.

Using the Workspace in process debugging mode

If you are in process debugging mode, the Workspace prompt changes to indicate the mode. Once in process debugging mode, you enter ObjectCenter commands in the same way as you would in component debugging mode. For example:

```
    pdm -> step
```

Like component debugging mode, in process debugging mode you can enter either ObjectCenter commands or code directly in the Workspace.

However, in process debugging mode, the Workspace does not give you access to the ObjectCenter interpreter. This means that, unlike component debugging mode where you can immediately execute any C++ or C statement from the Workspace, in process debugging mode you can only evaluate a limited range of C++ or C statements.

Furthermore, the two language modes (C++ mode and C mode) for the Workspace only apply to component debugging mode. Basically, in process debugging mode, you can evaluate any C++ or C expression at the Workspace that you could use as an argument to the **print** command. For more information, see the **print** and **pdm** entries in the *ObjectCenter Reference*.

Debugging information available

The debugging 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 were linked to create the executable. For more information on debugging information, see the **debug** entry in the *ObjectCenter Reference*.

Chapter 7 Visualizing your code

With ObjectCenter you can visualize and, thereby, comprehend code more quickly. ObjectCenter can be especially useful when you have to debug, enhance, or continue developing code others have written. It can reduce the time it takes to reacquaint yourself with code you wrote before. It can also speed up the development of new code.

This chapter describes the following tools that help you to visualize code:

- *Project Browser*
- *Inheritance Browser*
- *Class Examiner*
- *Cross-Reference Browser*
- *Data Browser*
- *Expand Command*
- *Pop-up menu information and Workspace commands*

Project Browser

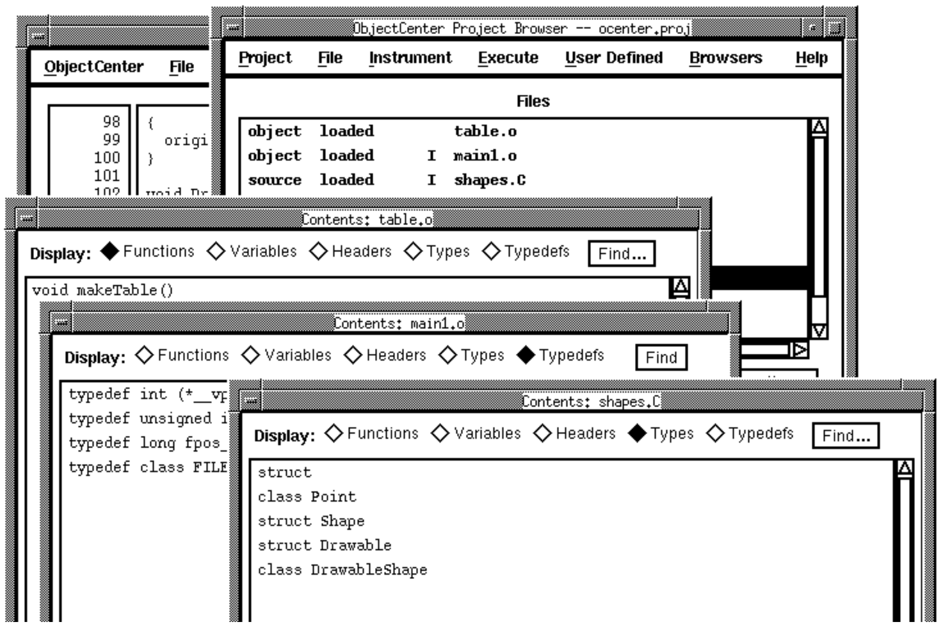
You use the Project Browser to get an overview of all the source, object, and library files that are the components of your current project. You use the File Contents and Library Contents windows of the Project Browser to examine the definitions of the functions, variables, headers, types, and typedefs contained in your files. Using the Project Browser to get an overview of the files in your project is available only in component debugging mode.

The Contents windows

To examine the functions, variables, headers, types, and typedefs defined in a file or library in your current project, you use the Contents windows of the Project Browser. These are the File Contents window and the Library Contents window.

The File Contents window

To open the File Contents window for any files listed in the Project Browser, select any files of interest in the Files area and then select the **Contents** button in the Files area. A File Contents window opens for each selected file. To view a particular kind of definition (functions, variables, headers, types, or typedefs) in a file, select the appropriate category from the display filter as shown in the illustration:



```
object  loaded      table.o
object  loaded    I main1.o
source  loaded    I shapes.C
```

```
void makeTable()
```

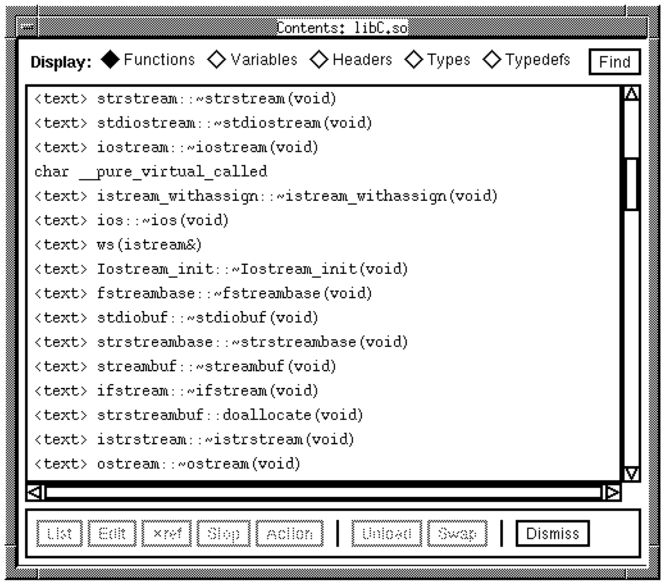
```
typedef int (*__vp
typedef unsigned i
typedef long fpos_
typedef class FILE
```

```
struct
class Point
struct Shape
struct Drawable
class DrawableShape
```

For header files loaded as source code, the File Contents window shows nesting of header files with indentation.

The Library Contents window

To open a Library Contents window for any library listed in the Project Browser, select any libraries of interest in the Libraries area and then select the **Contents** button in the Libraries area. A Library Contents window opens for each selected library. Here is an illustration:


Contents: libC.so
Display: ◆ Functions ◇ Variables ◇ Headers ◇ Types ◇ Typedefs
Find
List
Edit
xref
Stop
Action
Unload
Swap
Dismiss

```
<text> strstream::~strstream(void)
<text> stdiostream::~stdiostream(void)
<text> iostream::~iostream(void)
char __pure_virtual_called
<text> istream_withassign::~istream_withassign(void)
<text> ios::~ios(void)
<text> ws(istream&)
<text> Iostream_init::~Iostream_init(void)
<text> fstreambase::~fstreambase(void)
<text> stdiobuf::~stdiobuf(void)
<text> strstreambase::~strstreambase(void)
<text> streambuf::~streambuf(void)
<text> ifstream::~ifstream(void)
<text> strstreambuf::doallocate(void)
<text> istrstream::~istrstream(void)
<text> ostream::~ostream(void)
```

NOTE As shown above, shared libraries often contain a very large number of definitions. Displaying them can take an excessive amount of time. Therefore, we recommend you ordinarily display only static libraries.

The Library Members area lists the individual object files making up the library. To view the contents of a library member, select that object file in the Library Members area. (If the library you have selected is a PIC file, the Library Contents window only lists those library modules that are currently linked in.) To view a particular kind of definition (functions, variables, headers, types, or typedefs) in a library, select the appropriate category from the display filter.

Type information

To have type information, you must compile object files *with* the **-g** switch and load them into ObjectCenter *without* the **-G** switch. If an object file does not contain type information for an item, the File Contents and Library windows list the area in the object file in which the item is located: **<text>** or **<data>**. If the object file does not contain enough information even to determine the location of the item, then the type is listed as **<extern>**.

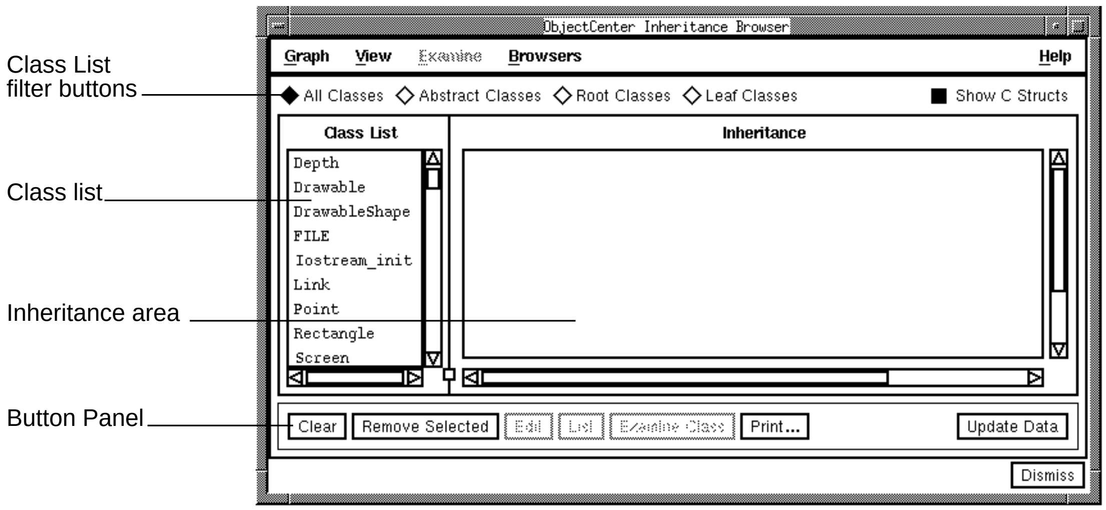
Inheritance Browser

The Inheritance Browser is one of the two class windows. The Inheritance Browser lists the names of the classes in the files you have loaded and displays a graphic representation of the inheritance relationships they have. The Class Examiner, the other class window, displays information about class members. See the 'Class Examiner' on page 186.

Accessing the Inheritance Browser

When you select **Inheritance Browser** in the **Browsers** menu, the Inheritance Browser appears.

This is what the Inheritance Browser looks like:



Class List

When the Inheritance Browser appears, the Class List contains an alphabetical list of the names of all the classes in the files you have loaded, including instantiated templates and C structs.

You can exclude names from the Class List by selecting the Class List filter button corresponding to the type of class you want to exclude. You can include the classes by unselecting the Class List filter button.

These are the types of classes you can use the Class List filter buttons to include or exclude:

- Abstract classes

  Abstract classes are classes with one or more pure virtual functions.

- Root classes

  Root classes are classes that have no base classes.

- Leaf classes

  Leaf classes are classes from which no other classes are derived.

- C structs

  C structs are classes that have no constructors, destructors, member functions, or base classes.

The **All Classes** filter button applies to C structs if the **Show C Structs** filter button is selected. If the **Show C Structs** filter button is unselected, the **All Class** filter button causes all classes except C structs to appear.

You can scroll through the names in the Class List using its vertical scrollbar. Some class names may be too long to appear in their entirety. To see more of the names too long to fit, use the horizontal scrollbar.

Inheritance area

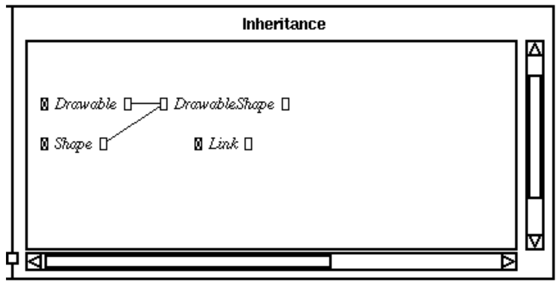
The Inheritance area is where you display the classes whose inheritance relationships you want to see. The area is blank when you access the Inheritance Browser from the Main Window. When you access it from a window requiring you to specify a class first, the specified class appears in the center of the Inheritance area.

Seeing how particular classes relate

You can quickly see the inheritance relationships of any classes you are interested in. To find out about inheritance relationships among certain classes, select their names in the Class List. As you select each class name, it appears in the Inheritance area. You can display as many classes as you want in the Inheritance area by selecting their names in the Class List. The Inheritance Browser automatically shows you whether an inheritance relationship exists between any two classes in any of the classes whose names you selected.

Another way you can see how classes relate is by using pointer boxes. See the 'Showing inheritance levels with pointer boxes' on page 182.

The following illustration shows the Inheritance area with three classes selected in the Class List. A line connects **Drawable** and **DrawableShape** because they have a parent-child relationship. **Link** is unconnected to **Drawable** and **DrawableShape**. It does not have a parent-child relationship with either one of the other two classes.



Selecting and unselecting names in the Inheritance area

You must select class names in the Inheritance area before you can do such things as display more inheritance levels and move and remove class names from the Inheritance area.

Selecting names

You can select class names in the Inheritance area with the mouse, the pop-up menu, or the Main Menu. Once you select a class name, a box appears around the class name.

In addition to selecting items individually, you can select all items in a rectangular area by dragging the mouse pointer to enclose the desired items. You can also select them by specifying a scope from the pop-up menu:

1. Move the mouse pointer to the item and display the pop-up menu with the Right mouse button.
2. From the pop-up menu, select **Select**.
3. From the submenu, select the scope you want. (The **Ancestors**, **Parents**, **Children**, and **Descendents** scopes only apply to items linked by pointers.)

Unselecting names

You can unselect class names in the Inheritance area with the mouse, the pop-up menu, or the Main Menu. Once you unselect a class name, the box around the class name disappears.

Accelerators for selecting and unselecting

The Inheritance Browser provides accelerator keys for selecting and unselecting class names in the Inheritance area. Table 20 contains the accelerator keys and what they select or unselect.

Table 20 Accelerator Keys: Selecting and Unselecting

Accelerator Keys	What They Do
Control + Extend mouse button	Select descendants
Control + Shift + Extend mouse button	Add descendants to selection
Control + Meta + Shift + Extend mouse button	Toggle selection state of descendants
A	Unselect descendants
Control + Select mouse button	Select ancestors
Control + Shift + Select mouse button	Add ancestors to selection
Control + Meta + Shift + Select mouse button	Toggle selection state of ancestors
C	Unselect ancestors

Showing inheritance levels

You can show the inheritance levels of any classes you select. To show inheritance levels, you can use the Main Menu or the pop-up menu.

To show levels of inheritance using the pop-up menu:

1. Put the cursor on the name of the class whose inheritance you want to show. The name of the class does not have to be selected in the Inheritance area.
2. Choose **Show** on the pop-up menu.
3. Cascade into **Children**, **Descendants**, **Parents,** or **Ancestors**, depending on what you want to show.

   The class names on the branch you selected appear in the Inheritance area, and they become selected in the Class List.

Showing inheritance levels with pointer boxes

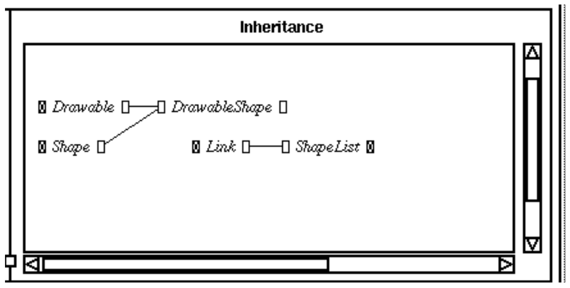
You can show inheritance levels with pointer boxes. A pointer box appears left of and right of each class name in the Inheritance area.

The pointer box to the right of the class name in the Inheritance area refers to the children of that class. If the pointer box is empty, the class has at least one child. If the pointer box has an "X", the class has no children.

The pointer box to the left of a class name refers to the parents of that class. If the pointer box is empty, the class has at least one parent. If the pointer box has an "X", the class has no parents.

To see the parents of any class, click on the pointer to the left of its name. To see the children of any class, click on the pointer to the right of its name. Parents and children that now appear in the Inheritance area also appear selected in the Class List. You can show as many levels of inheritance as you want.

In the previous example, **DrawableShape** and **Link** have empty pointer boxes to the right of their names, indicating that they have at least one child each. Here is the Inheritance area now after clicking on the empty pointer box to the right of **Link**. **ShapeList**, the only child of **Link**, appears and is selected in the Class List.



Displaying more of the Inheritance area

You may want to see parts of the inheritance hierarchy that don’t fit in the Inheritance area.

To display another part of the inheritance hierarchy, use the horizontal and vertical scrollbars of the Inheritance area.

To display as much of the hierarchy in the Inheritance area as you want at any one time, resize the entire Inheritance Browser.

As an alternative to using scrollbars, you can customize the Inheritance Browser to display a canvas representing the virtual display area and a panner that represents what is currently shown in the Reference area.

For more information on using X resources to customize the ObjectCenter GUI, see the **X resources** entry in the *ObjectCenter Reference.*

Moving names of classes within the display

You may want to change the position of classes in the Inheritance area to reflect your visualization of the inheritance hierarchies you want to analyze. Once a class name is in the Inheritance area, you can drag it anywhere in the Inheritance area. If you want to move several class names as a group, select them first. Put the cursor on any one of their names and drag them as a group.

When you move a class name that is connected to a parent or child by a line, a line still connects them after the move. The line becomes longer or shorter, depending on the resulting distance between them.

When you move a parent to the right of its child, the line becomes dashed. When you move a child to the left of its parent, the line becomes dashed.

Clearing the Inheritance area

To clear the Inheritance area of all class names, do one of the following:

- Press the **Clear** button on the Button panel, or
- Choose **Clear** in the **Graph** menu of the Inheritance Browser**.**

All the class names in the display disappear, and they become unselected in the Class List.

Removing selected class names from the Inheritance area

You can remove selected class names from the Inheritance area with the Class List or in the Inheritance area.

To remove class names from the Inheritance area individually, unselect their names in the Class List. The class names disappear from the Inheritance area as you unselect them in the Class List.

You can also remove class names from the Inheritance area with the Button panel, pop-up menu, Main Menu, or the Delete key.

To remove class names with the Button panel:

1. Select the class name or names you want to remove.
2. Click on the **Remove Selected** button on the Button panel. The class names disappear, and they become unselected in the Class List.

Removing unselected class names from the Inheritance area

In addition to removing selected class names, you can keep unselected branches in the Inheritance area:

1. Select the class names you want to keep.
2. Select **Remove Unselected** in the **Graph** menu of the Inheritance Browser**.** Class names that were unselected disappear, and they are unselected in the Class List.

Listing the code that defines a class

You can list the source code that defines a class by selecting its name in the Inheritance area and using the Main Menu or the Button panel. Without having selected the name of a class, you can also use the pop-up menu.

To list the code by using the pop-up menu:

1. Put the cursor in the Inheritance area on the name of the class whose code you want to list. The class does not have to be selected in the Inheritance area.
2. Choose **List** on the pop-up menu. The code appears in the Source area of the Main Window.

Editing the code that defines a class

You can edit the source code that defines a class by selecting its name in the Inheritance area and using the Main Menu or the Button panel. Without having selected the name of a class, you can also use the pop-up menu.

To edit the code by using the pop-up menu:

1. Put the cursor in the Inheritance area on the name of the class whose definition you want to edit. The class does not have to be selected in the Inheritance area.
2. Choose **Edit** on the pop-up menu. The Inheritance Browser has accessed your editor, displaying the lines of code containing the definition you want to edit.

Updating the Inheritance Browser

The Inheritance Browser displays a message when ObjectCenter senses that the reference information for the Inheritance Browser may be out-of-date. Loading, reloading, unloading, or swapping a file can cause the reference information in the Inheritance Browser to be out-of-date.

If the message appears, click on **Update** to ensure that the reference information available to the Inheritance Browser is up-to-date with the current state of your project.

Examining the members of a particular class

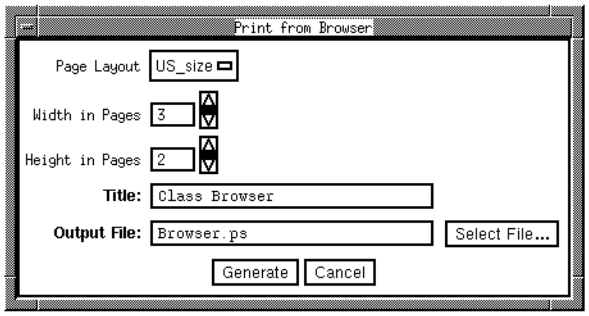
You can examine the members of a class by accessing the Class Examiner. See the 'Class Examiner' on page 186. You can access the Class Examiner by selecting the name of a class and using the Main Menu or the Button panel. You can also use the pop-up menu without having selected the name of a class.

To access the Class Examiner with the pop-up menu:

1. Put the cursor in the Inheritance area on the name of the class whose members you want to examine. The class does not have to be selected.
2. Choose **Examine Class** from the pop-up menu. The Class Examiner appears, focusing on the class you selected.

Postscript printing from the Inheritance Browser

You can print the contents of the Inheritance Browser to a Postscript file. Click on the **Print...** button in the Browser to see a dialog box in which you specify what paper size to use, the title of the printout, and the name and location of the output file. The default location is your current directory. You can also specify how many pages the output will be printed on. For example, if you specify an output page width of 3 and height of 2, the output is resized to fit on six sheets of paper.


Print from Browser

Page Layout: US_size

Width in Pages: 3

Height in Pages: 2

Title: Class Browser

Output File: Browser.ps

Finding more information

For more information, see the **classinfo** entry in the *ObjectCenter Reference.*

Class Examiner

The Class Examiner is one of the two class windows. The Class Examiner displays information about class members. The Inheritance Browser, the other class window, displays a graphic representation of the inheritance relationships among classes. See the 'Inheritance Browser' on page 178.

Accessing the Class Examiner

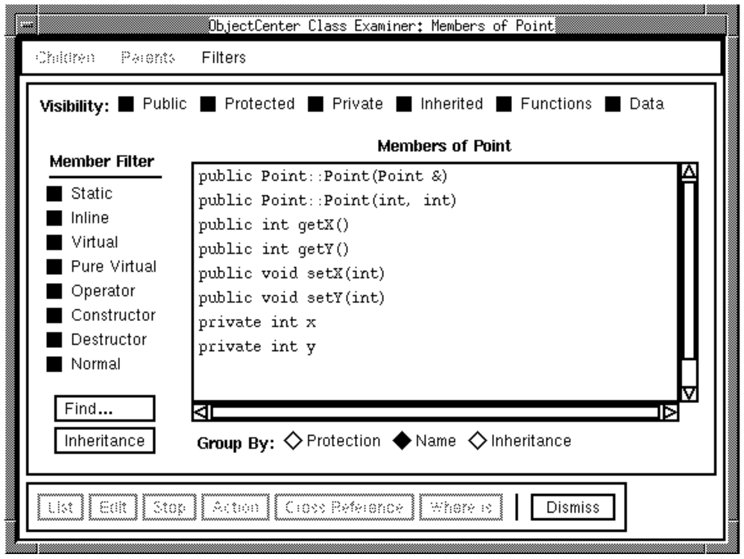
You can access the Class Examiner from the Main Window with or without having selected the name of the class you are interested in, in the Source area.

If the name of the class you want to examine appears in the Source area, follow these steps to access the Class Examiner:

1. Select the class name in the Source area or in the Workspace.
2. Do one of the following: Click on **Examine Class...** in the Button panel or choose **Examine Class** in the Examine menu in the Main Window. The Class Examiner appears.

If the name of the class you want to examine does not appear in the Source area, you must enter the class name in a dialog box that appears after you make an examine-class choice in the GUI. You select **Browse** in the dialog box, and the Class Examiner appears.

This is what the Class Examiner looks like:



```
public Point::Point(Point &)
public Point::Point(int, int)
public int getX()
public int getY()
public void setX(int)
public void setY(int)
private int x
private int y
```

In this example, the Class Examiner is focused on a class called **Point**.

When the Class Examiner appears, the Members area lists all the members of the focus class sorted by name.

You can scroll through the names in the Members area using the vertical scrollbar. Some member names may be too long to appear in their entirety. To see more of the names too long to fit, use the horizontal scrollbar.

Keeping member names visible

When the Class Examiner appears, it displays the names of all the members of the class. All of the Visibility buttons are selected.

If you are not interested in displaying the names of inherited class members, unselect the **Inherited** Visibility button. Inherited members disappear from the Members area.

If you want to remove the names of other kinds of class members from the Members area, unselect the **Public**, **Protected**, **Private**, **Function**, and **Data Visibility** buttons, depending on the kinds of class members you want the Members area to keep visible. A class member must meet all the selected characteristics to remain visible in the Members area.

Filtering member names

You can filter out names of certain kinds of class members by using the Member Filter buttons. These are the kinds of class members that you can filter out:

- Static
- Inline
- Virtual
- Pure Virtual
- Operator
- Constructor
- Destructor
- Normal

Normal members are members that belong to no other of the categories above.

You can select any one or any combination of Member Filter buttons. The Members area displays a class member if it meets at least one of the characteristics selected. To limit the display to names of class members with only one of the characteristics, the rest of the characteristics must be unselected.

Grouping member names

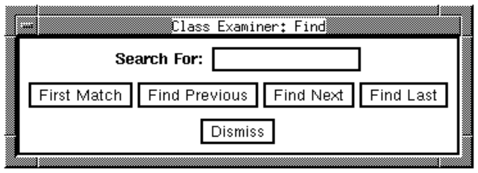
You can group class member names by protection level, name, or inheritance. To change the grouping, select the Group By button that corresponds to the grouping you want: **Protection**, **Name**, or **Inheritance**.

When you select one of these buttons, the class member names in the Members area are immediately sorted in the order you have selected. When you sort by **Protection** or **Inheritance**, the secondary sort is **Name**.

Searching for class members

You can search for class members in this way:

1 Click on the **Find** button. A dialog box appears:



2 In the **Search For** field, enter the name of the class member you would like to search for.

3 Click on **First Match** to see the first entry in the Members area that matches your search criterion, or click on **Find Previous** to see the previous match, **Find Next** to see the next match, and **Find Last** to see the last match in the Members area.

The search is limited to only those class members that are visible in the Members area. The search excludes class members you have filtered out or made to disappear from the Members area.

Finding the code that defines a member function

You can find the source code that defines a member function:

1 Select a text fragment in the name of the member function whose definition you want to find.

2 Click on the **Where is** button in the Button panel. The filename and line number of the source code that defines the member function appears.

Listing the code that defines a member function

You can list the source code that defines a member function:

1 Select the name of the member function whose definition you want to see.

2 Click on the **List** button in the Button panel. The source code that defines the member function appears.

Editing the code that defines a member function

You can edit the source code that defines a member class:

1 Select the name of the member function whose definition you want to see.

2 Click on the **Edit** button in the Button panel. The Class Examiner has accessed your editor, displaying the lines of code containing the definition of the member function you want to edit.

Choosing another class to examine

You can examine any child or parent of the class you are examining by clicking on the **Children** or **Parents** button and cascading to the submenu to select the name of the class you want to examine.

Accessing other windows from the Class Examiner

From the Class Examiner, you can access the Cross-Reference Browser by selecting the **Cross-Reference** button in the Button panel.To access the Inheritance Browser, select the **Inheritance** button.

Finding more information

For more information, see the **classinfo** entry in the *ObjectCenter Reference*.

Cross-Reference Browser

You use the Cross-Reference Browser to see a graphic representation of the calling structure of your program. It shows the references to and from functions and global variables. You can show the calling hierarchy at any depth and filter the display to suit your needs at any given moment. Using the Cross-Reference Browser to see the calling structure of your program is available only in component debugging mode.

Cross-referencing functions and global variables

If you cross-reference a function, the Cross-Reference Browser shows all of the functions where that function is called and all of the functions and global variables that the function references.

If you cross-reference a global variable, the Cross-Reference Browser shows all functions in which the variable is used and all other variables that use the named variable for an initialization value. If the named variable is initialized, all functions and variables it uses for its initialization value are also listed.

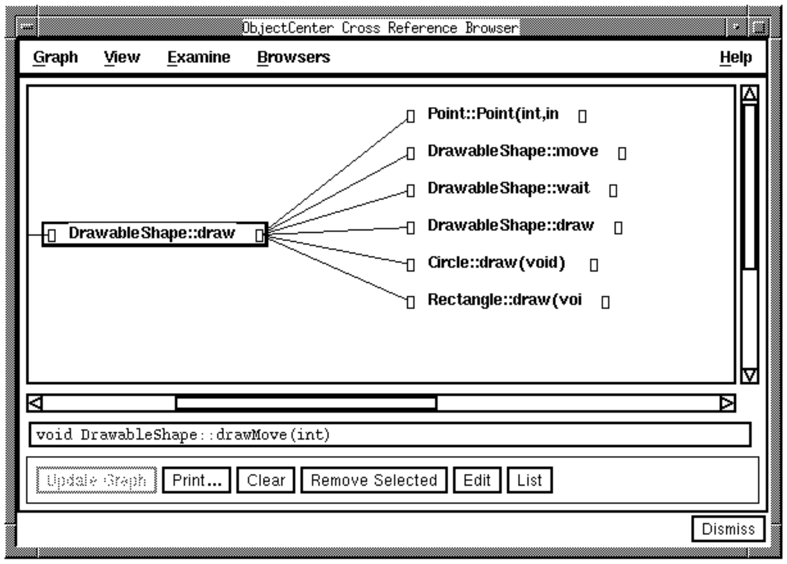
NOTE Cross-referencing functions or global variables in shared libraries is unreliable and changes depending on the execution state of your program. This is due to the way that symbols are linked from shared libraries. If you need accurate cross-reference information for a library, load and link it as a static library.

Displaying virtual functions

Whenever there is a reference to a virtual function, the Cross-Reference Browser displays each instance of the virtual function. In the following example, **drawMove()** has this definition:

```
void DrawableShape::drawMove(count)
{
  move(Point(...));
  draw();
  wait();
}
```

In this example, **draw()** is a virtual function implemented by the subclasses of **DrawableShape**. This is how **drawMove()** appears in the Cross-Reference Browser:



```
void DrawableShape::drawMove(int)
```

The Browser shows that there are three instances of **draw()**, implemented by the classes **DrawableShape**, **Circle**, and **Rectangle**.

NOTE The Cross-Reference Browser only recognizes static references between functions or global variables. If a reference was dynamically created, such as by assigning the address of a variable to a pointer, it is not shown.

Formal arguments, automatic variables, macros, and typedefs cannot be cross-referenced.

You specify a function or global variable to cross-reference in any of the ways listed in Table 21.

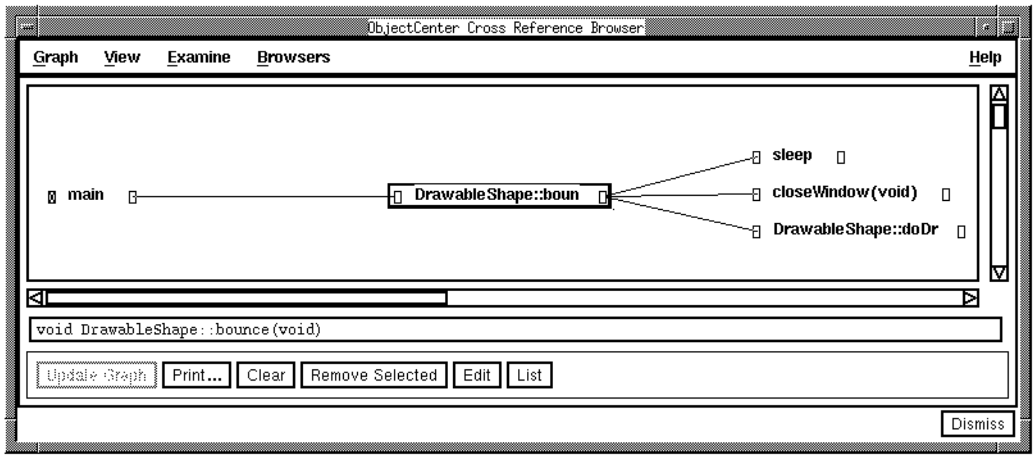
Table 21 Cross-Referencing a Function or Variable: Methods According to Work Area

Work Area	Ways to Cross-Reference a Function or Global Variable
Main Window	**1** From anywhere in ObjectCenter, use the mouse to select a function or global variable you want to display. **2** From the **Examine** menu, select **xref**.
Workspace	Use the **xref** command.

Using the Cross-Reference Browser

When you cross-reference a function or global variable, the Cross-Reference Browser opens and the Reference area displays the referencing structure of the functions or global variables you have specified.

For example, to examine all the locations where the function **DrawableShape::bounce()** is used and all the global symbols it uses, select **bounce** and invoke the Cross-Reference Browser as shown in this illustration:



void DrawableShape::bounce(void)

In this example, **bounce()** is called only by **main()**, and it calls three functions: **sleep()**, **closeWindow()**, and **DrawableShape::doDraw()**.

Interpreting reference lines

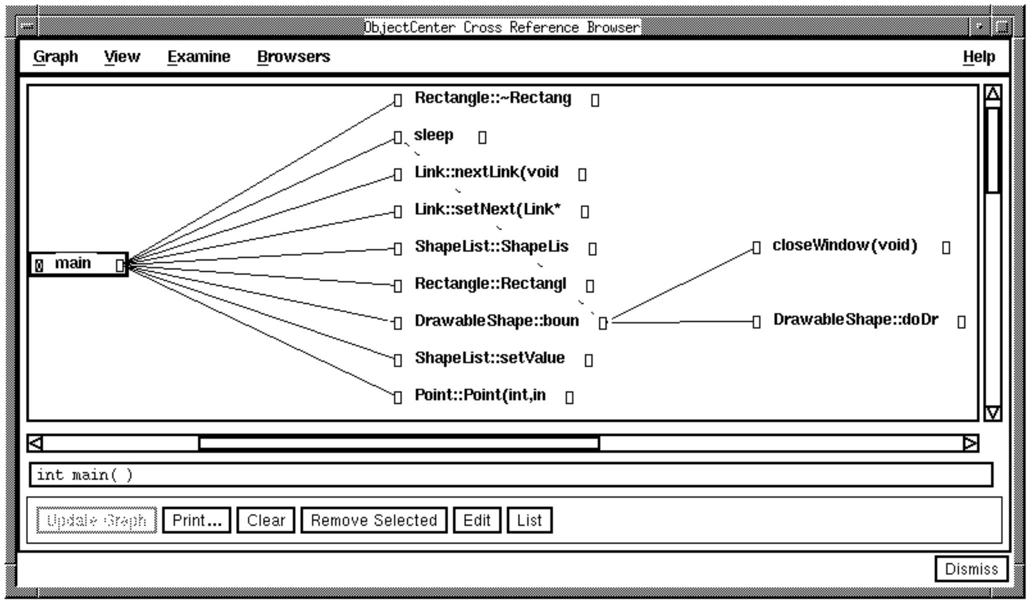
When a reference between two items is shown, it is represented by a line connecting the right reference box of one item with the left reference box of another item. Typically, an item making a reference is to the left of the item receiving the reference.

As an aid for following references that are displayed in the opposite direction (right to left), the Reference area uses a dashed line. That is, if the item making the reference is to the right of the item receiving the reference, they are connected with a dashed line.

Showing further references and removing them

You can show additional references to or from an item by clicking on the reference box of that item. To remove items, you must select them (See 'Selecting and manipulating reference items' on page 195) and select the **Remove Selected** button.

In the previous example, **bounce()** is the focus item and **main()** is shown as referencing it. To show all the other references made by **main()**, click on the reference box at the right of **main()**.



Navigating in the Reference area

You can display as many functions or global variables as you want in the Cross-Reference Browser. To move a reference item into view in the Reference area, use the horizontal and vertical scrollbars.

As an alternative to using scrollbars, you can customize the Cross-Reference Browser to display a canvas representing the virtual display area and a panner that represents what is currently shown in the Reference area. For more information on using X resources to customize the ObjectCenter GUI, see the **X resources** entry in the *ObjectCenter Reference.*

Updating the Reference area

The Cross-Reference Browser displays a message when the reference information might be out-of-date. Loading, reloading, unloading, or swapping a file can cause the reference information in the Cross-Reference Browser to be out-of-date.

If the message appears, click on **Update Graph** to ensure that the reference information available to the Cross-Reference Browser is up-to-date with the current state of your project.

Selecting and manipulating reference items

Each function or global variable shown in the Cross-Reference Browser is treated as a separate item and can be selected or unselected individually or as a group.

In addition to selecting items individually, you can select all the items in a rectangular area by dragging the mouse pointer to enclose the desired items. You can also select items by specifying a scope from the pop-up menu in the Reference area as follows:

1. Move the mouse pointer to the reference item and display the pop-up menu with the Right mouse button.
2. From the pop-up menu, choose **Select**.
3. From the submenu, select the scope you want. (The **Ancestors**, **Parents**, **Children**, and **Descendents** scopes only apply to items linked by pointers.)

   A child is a function called from another function. A parent is a function from which another function is called. Descendants are the children of a function, their children, and so on. Ancestors are the parents of a function, their parents, and so on.

Once a file or a group of files is selected, you can move it around the Reference area or use the pop-up menu to remove it from the Cross-Reference Browser.

Changing the number of characters displayed

By default, the Cross-Reference Browser displays a maximum of 29 characters in each function cell. If you want to change the number of characters displayed, edit your .**Xdefaults** file by adding this line:

```
ObjectCenter*XrefBrowser.XrefColumnWidth: num_of_char
```

For *num_of_char*, enter the maximum number of characters you want displayed in function cells. For more information about X11 resources that you can modify to customize ObjectCenter, see the **X resources** entry in the *ObjectCenter Reference*.

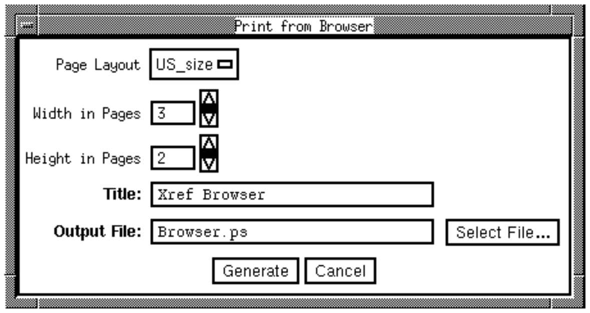
Displaying the return type

By default, the Cross-Reference Browser does not display the return type of a function in its cell. If you want the Cross-Reference Browser to display the return type, edit your .**Xdefaults** file by adding this line:

```
ObjectCenter*XrefBrowser*showReturnType:    True
```

Postscript printing from the Cross-Reference Browser

You can print the contents of the Cross-Reference Browser to a Postscript file. Click on the **Print...** button in the Browser to see a dialog box in which you specify what paper size to use, the title of the printout, and the name and location of the output file. The default location is your current directory. You can also specify how many pages the output will be printed on. For example, if you specify an output page width of 3 and height of 2, the output is resized to fit on six sheets of paper.


Print from Browser
Page Layout: US_size
Width in Pages: 3
Height in Pages: 2
Title: Xref Browser
Output File: Browser.ps

Finding more information

For more information, see the **xref** entry in the *ObjectCenter Reference*.

Data Browser

You use the Data Browser to display a graphical representation of any data structure. The Data Browser has an especially strong connection with interactive debugging in either component or process debugging mode. Typically, you use the Data Browser when you are at a break level and want to investigate the state of some data structure that is currently in scope. You may also examine global data at any time.

Displaying data structures

Table 22 lists all of the ways to display a data structure.

Table 22 Displaying a Data Structure: Methods According to Work Area

Work Area	Ways to Display a Data Structure
Main Window	**1** From anywhere in ObjectCenter, use the mouse to select a variable you want to display. The variable must currently be in scope. **2** From the **Examine** menu, select **display**.
Data Browser	**1** From the **Graph** menu, select **New Expression**. **2** In the dialog box, enter the variable or expression at the input line. **3** Select the **Data Browse** button.
Workspace	Use the **display** command.

Using the Data Browser

When you display data structures, the Data Browser opens and the Data area displays a graphic representation for the data structures you have specified.

Changing values for data structure elements

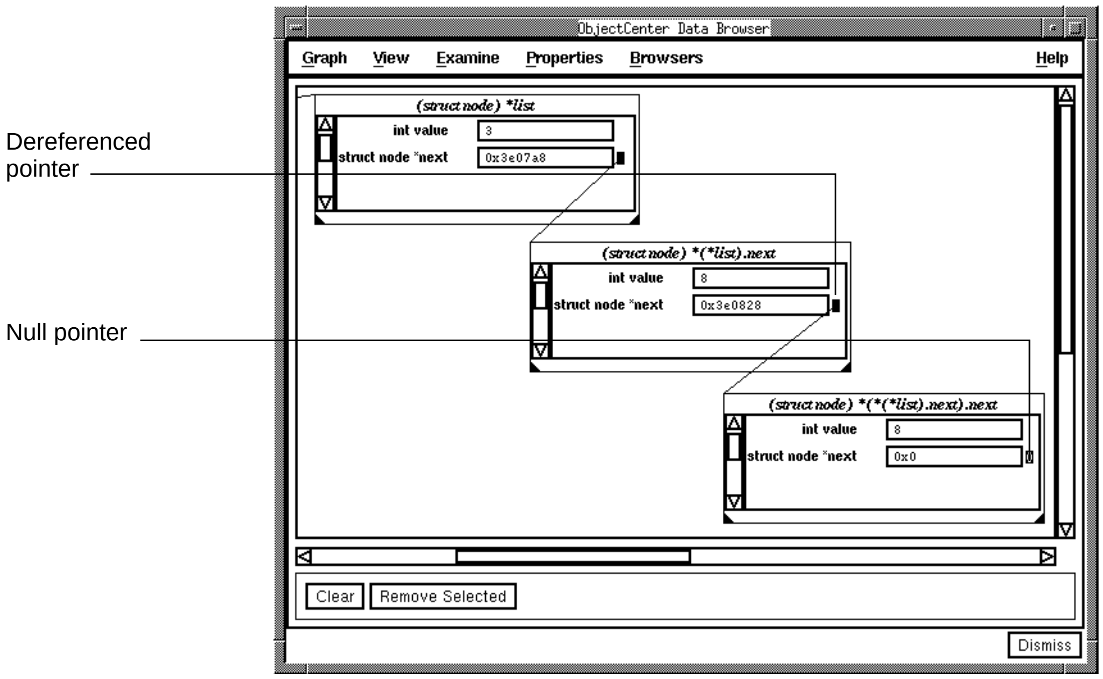
Complex data structures like arrays and structures also have a scrolling list for elements in the structure folders you can open to see all the elements of the structure.

When you are in component debugging mode, you can change the values for structure elements directly in the Data area. Select the Value Field you want to change and enter a new value.

When you display the Data Browser in process debugging mode, you cannot use it to change the values in the elements of data structures as you can in component debugging mode.

Dereferencing pointers and following linked lists

Values that are pointers have an empty pointer box to the right of the pointer value. Clicking on the empty box dereferences the pointer and displays the item to which it points. The Data Browser draws a line between the box and the new data item in the linked list. If the data in the item contains an invalid pointer, the pointer box contains an "X" instead of being empty.



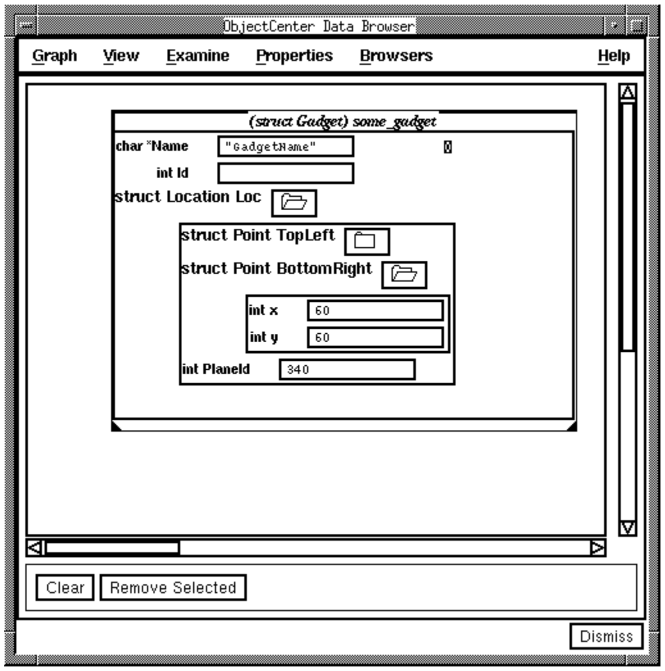
(struct node) *list
int value 3
struct node *next 0x3e07a8

(struct node) *(*list).next
int value 8
struct node *next 0x3e0828

(struct node) *(*(*list).next).next
int value 8
struct node *next 0x0

Structures appear in data elements as folders, which you can choose to open by clicking on them. In the next example, **some_gadget** is an instance of the structure **Gadget**, which contains **Loc**, an instance of the structure **Location**.

The folder for **Location** is open to display its contents, which include two instances of the structure **Point**. The folder for **TopLeft**, one instance of the structure **Point**, is closed. The folder for **BottomRight**, another instance of **Point**, is open to show its contents: **int x** and **int y.**



Removing items

To remove data items, you must select them (See 'Selecting and manipulating data items' on page 200) and select the **Remove Selected** button.

Interpreting reference lines

The Data Browser draws a solid line if the pointer points to the top of a data element. The deference symbol may not actually be visible in the Data Browser because it may be:

- In an array or structure and has scrolled off the list
- In a nested array or structure
- Hidden with property sheet hiding

If the reason the deference symbol is not visible is that it is in an array or structure and has scrolled off the list, the solid line starts at the corner of the dereference symbol box. If the reason the dereference symbol is not visible is that it is in a nested array or hidden with property sheet hiding, then the solid line starts between the corners.

The Data Browser draws a dotted line if the pointer points to an element other than the first element in the structure. Even though the pointer points to an element other than the first, the dotted line touches the top corner of the item.

Navigating in the Data area

You can display as many data structures as you want in the Data Browser. To move a display item into view in the Data area, use the horizontal and vertical scrollbars.

As an alternative to using scrollbars, you can customize the Data Browser to display a canvas representing the virtual display area and a panner that represents what is currently shown in the Data area.

For information on using X resources to customize the ObjectCenter GUI, see the **X resources** entry in the *ObjectCenter Reference*.

Updating data items

When data has changed, data items in the Browser are automatically updated at every break in execution and at every assignment statement in the Workspace.

TIP: Improving performance when stepping through code

If you are stepping through your code and have many data items displayed in the Data Browser, the time spent in updating the Data area at each break in execution can degrade performance. You can improve performance by minimizing the number of items you have displayed or dismissing the Data Browser.

Selecting and manipulating data items

Each data structure shown in the Data Browser is treated as a separate item and can be selected or unselected individually or as a group.

In addition to selecting items individually, you can select all items in a rectangular area by dragging the mouse pointer to enclose the desired items.

You can also select them by specifying a scope from the pop-up menu in the Data area in the following way:

1. Move the mouse pointer to the reference item and display the pop-up menu with the Right mouse button.
2. From the pop-up menu, choose **Select**.
3. From the submenu, select the scope you want. (The **Ancestors**, **Parents**, **Children**, and **Descendents** scopes only apply to items linked by pointers.)

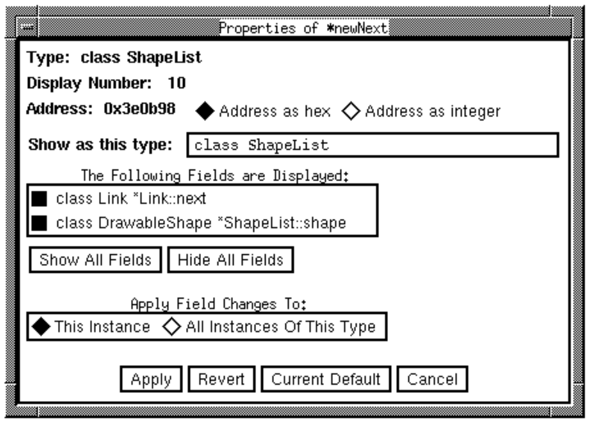
Once an item or a group of items is selected, you can use the pop-up menu to remove them from the Data Browser or use the **Remove Selected** button.

You can also move selected items by dragging them where you want them.

Changing display properties

You can change the display of a data structure. Changing the display affects only the display of the data structure in the Data Browser, not the definition of the data structure itself.

One of the ways you can change the display of a data structure is with the Properties dialog box. Select a data structure and select **Properties/Item Properties**. The Properties dialog box appears showing the type, display number of the item and its address. Here is what the Properties dialog box looks like:



Type: class ShapeList
Display Number: 10
Address: 0x3e0b98 [✓ Address as hex] Address as integer
Show as this type: class ShapeList
The Following Fields are Displayed:
■ class Link *Link::next
■ class DrawableShape *ShapeList::shape

Apply Field Changes To:
[✓ This Instance] All Instances Of This Type

In the Properties dialog box, you can choose to:

- Display the address in hexadecimal notation or as an integer.
- Display the type casting you choose.
- Display all or none of the fields of the data structure, or limit the display to only the specific fields of the data structure you are interested in.
- Apply display changes only to the one data item you selected or to all the data items of its type.

Once you make changes in the Properties dialog box, you can select:

- **Apply** to apply the changes you set.
- **Revert** to revert the settings to the way they were when you opened the box, unless you selected **Apply** since you opened the box. If you selected **Apply** since you opened the box, then the settings revert to the way they were when you selected **Apply**.
- **Current Default** to use the global display settings for data of that type.
- **Cancel** to close the box.

You can change the display of a selected data structure outside the Properties dialog window by selecting **Properties/View...** from the main menu of the Data Browser and then selecting one of the following:

- **Iconify** if you want to keep the data structure displayed but want to save space by hiding all but its name. Select **Deiconify** to revert to the display before you selected **iconify**.
- **Raise** if you want to shuffle the data structure up from one that was covering it. Select **Lower** if you want to shuffle it down to uncover one that was hidden.
- **Shrink** if you want to close all the folders of all the fields of the data structure. Select **Zoom** if you want to open all the folders and make the data item expand to show all items.

Finding more information

For more information, see the **display** entry in the *ObjectCenter Reference.*

Expanding C++ statements

It is often difficult to see exactly what code needs to be executed in order to carry out a particular C++ statement. For example, your code might call overloaded operators and functions or a user-defined conversion. None of these calls are obvious by simply looking at the code.

Using the expand command

ObjectCenter provides several ways for you to better understand the flow of your code. One of the most powerful is the **expand** command.

The **expand** command allows you to see which functions could possibly be called if a section of code is executed. (The set of functions *actually* executed during a given run might depend on run-time conditions and so might be a subset of the functions listed by **expand**.) Expanding a statement allows you to see implicit function calls and disambiguates overloaded functions and operators.

To expand C++ statements, first select a section of code in the Source area, then choose **expand** in the **Examine** menu in the Main Window.

What expand does

When you issue **expand**, ObjectCenter searches for statements contained in the selected text:

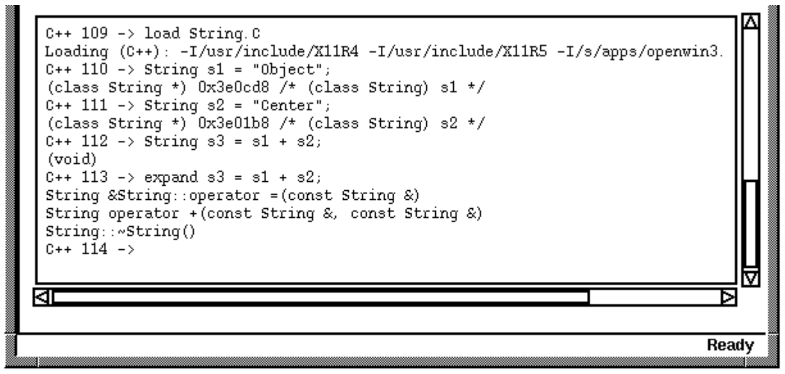
- If no complete statement is contained in the selected code, or if there is no function call in the selected code, ObjectCenter takes no action when you issue **expand**.
- If there are function calls in the selected statements, ObjectCenter opens a pop-up window and lists the functions that could be called, including user-defined conversions.

Note that the **expand** command *evaluates* the selected statements; it does not execute them. No program functions are actually called and no program values are changed by the evaluation of the statements.

Two examples

NOTE If any class objects were constructed in the code you selected, the **expand** command shows at the end the destructors that would be called when the objects went out of scope.

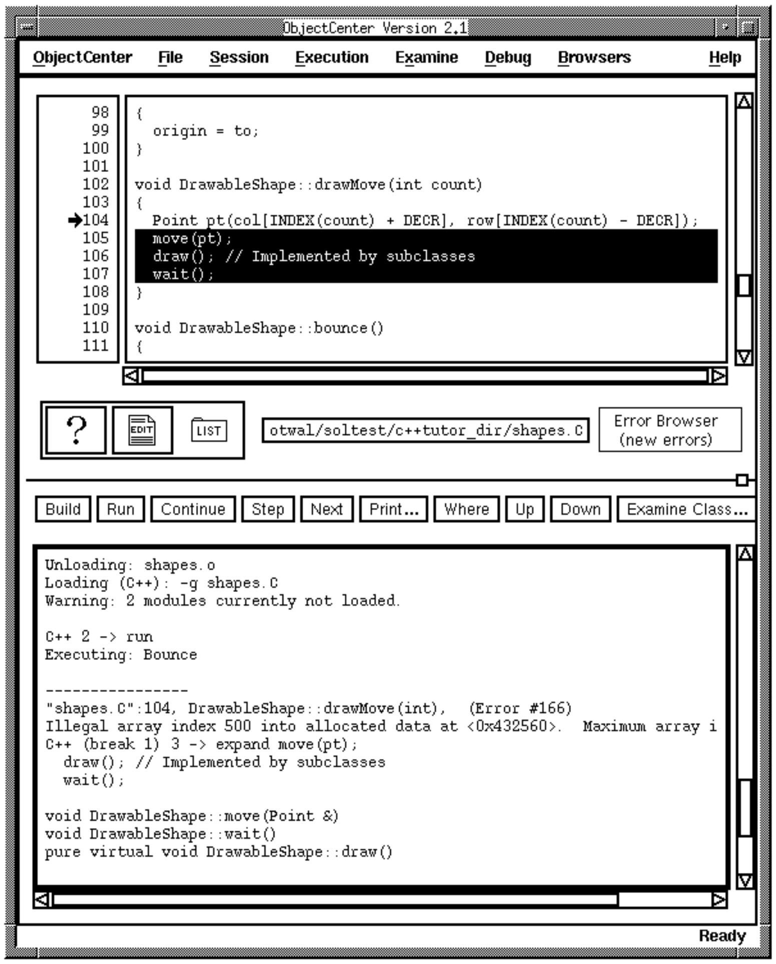
In the following example, **s1, s2**, and **s3** are all instances of the **String** class. The statement **s3 = s1** + **s2** is selected and evaluated using the **expand** command in the Workspace.



```
C++ 109 -> load String.C
Loading (C++): -I/usr/include/X11R4 -I/usr/include/X11R5 -I/s/apps/openwin3.
C++ 110 -> String s1 = "Object";
(class String *) 0x3e0cd8 /* (class String) s1 */
C++ 111 -> String s2 = "Center";
(class String *) 0x3e01b8 /* (class String) s2 */
C++ 112 -> String s3 = s1 + s2;
(void)
C++ 113 -> expand s3 = s1 + s2;
String &String::operator =(const String &)
String operator +(const String &, const String &)
String::~String()
C++ 114 ->
```

As shown, this statement involves three function calls: two overloaded operators (operator= and operator+) and a **String** destructor.

You can also select several statements to expand. In addition, you can use the **expand** choice in the **Examine** menu. In the example in the next illustration, selecting **expand** from the **Examine** menu results in the display of the functions that the lines of selected code call.



```
{
  origin = to;
}

void DrawableShape::drawMove(int count)
{
  Point pt(col[INDEX(count) + DECR], row[INDEX(count) - DECR]);
  move(pt);
  draw(); // Implemented by subclasses
  wait();
}

void DrawableShape::bounce()
{
```

```
Unloading: shapes.o
Loading (C++): -g shapes.C
Warning: 2 modules currently not loaded.

C++ 2 -> run
Executing: Bounce

----------------
"shapes.C":104, DrawableShape::drawMove(int),   (Error #166)
Illegal array index 500 into allocated data at <0x432560>.  Maximum array i
C++ (break 1) 3 -> expand move(pt);
  draw(); // Implemented by subclasses
  wait();

void DrawableShape::move(Point &)
void DrawableShape::wait()
pure virtual void DrawableShape::draw()
```

Pop-up menu and Workspace commands

ObjectCenter offers a pop-up menu and pop-up information windows that help you understand your code better by showing information about selected identifiers or expressions listed in the Source area or Workspace.

Expressions Options (shift-right) menu

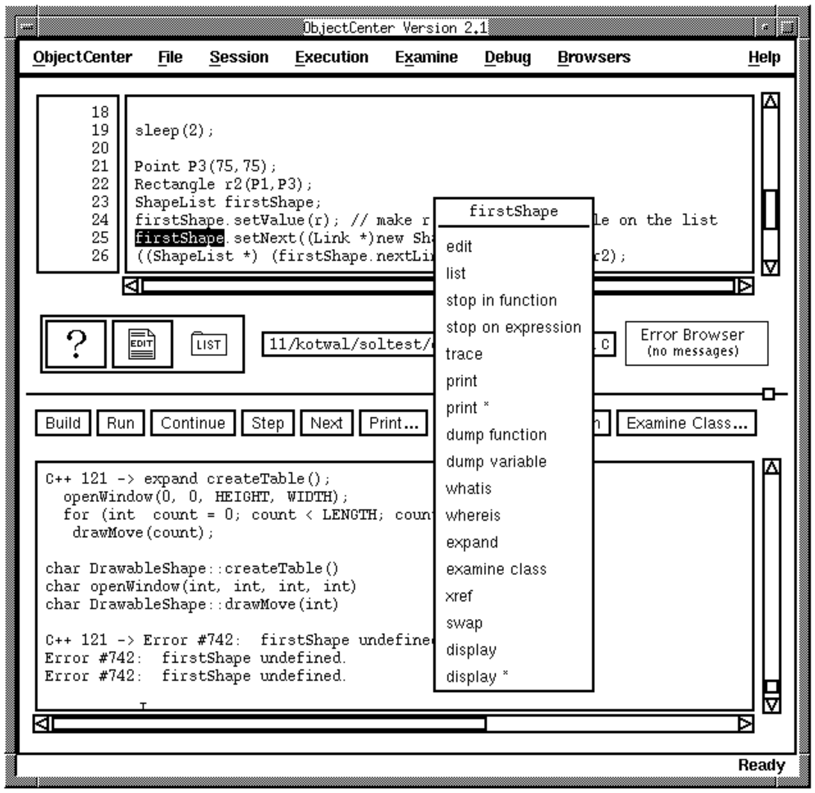
You can get information on an identifier or expression listed in the Source area by accessing the **Expressions Options** (shift-right) menu.

To access the **Expressions Options** menu:

1. Put the cursor on the expression you want information about. The expression can be either in the Source area or Workspace and can be either selected or unselected. If it is unselected, however, you must have no other expression selected. If another expression is selected, you must unselect it.
2. Press and hold down the Shift key. Then, click the Right mouse button.

The **Expressions Options** menu appears. The first line of the menu shows the expression you want information about.

Here is an illustration:

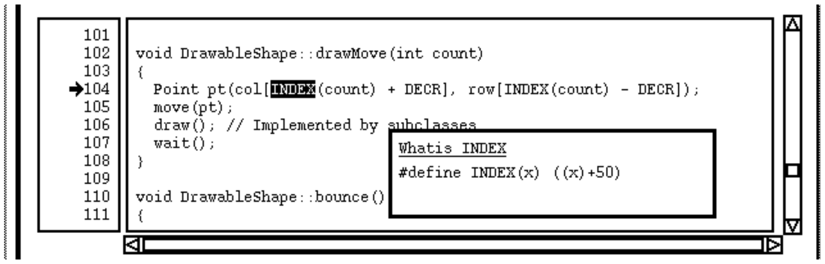


```
sleep(2);

Point P3(75,75);
Rectangle r2(P1,P3);
ShapeList firstShape;
firstShape.setValue(r); // make r                    le on the list
firstShape.setNext((Link *)new Sh
((ShapeList *) (firstShape.nextLi                    r2);
```

```
C++ 121 -> expand createTable();
  openWindow(0, 0, HEIGHT, WIDTH);
  for (int  count = 0; count < LENGTH; coun
   drawMove(count);

char DrawableShape::createTable()
char openWindow(int, int, int, int)
char DrawableShape::drawMove(int)

C++ 121 -> Error #742:  firstShape undefine
Error #742:  firstShape undefined.
Error #742:  firstShape undefined.
```

Select the command you want to apply to the expression. If you have selected one of the commands that opens a window of information about your code, that window remains in the screen until you click the Left mouse button.

For example, this illustration shows the **whatis** pop-up window.



```
void DrawableShape::drawMove(int count)
{
  Point pt(col[INDEX(count) + DECR], row[INDEX(count) - DECR]);
  move(pt);
  draw(); // Implemented by subclasses
  wait();
}

void DrawableShape::bounce()
{
```

```
Whatis INDEX
#define INDEX(x)  ((x)+50)
```

Table 23 contains the names of all the commands in the **Expressions Options** menu.

Table 23 Expressions Options Menu

Menu Selection	Description
Edit	Invokes your editor at the specified location.
List	Displays source code lines.
Stop in function	Sets a breakpoint at the first line of the function if you specify a function.
Stop on expression	Sets a breakpoint on the expression, if you specify an expression. Stops execution whenever the expression changes.
Trace	Displays each line of code as it is being executed. If the location is in a function, tracing is limited to within that function.
Print	Shows the value of the variable or expression.
Print*	Shows the dereferenced value of the pointer.
Dump function	Displays the name and value of each variable local to the specified function.
Dump variable	Displays the name and value of the variables in the text string.
Whatis	Displays all uses of a name for a function, data variable, tag name, enumerator, type definition, or macro definition.
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.

Table 23 Expressions Options Menu (Continued)

Menu Selection	Description
Expand	Displays which functions a given section of code could possibly call. For this **Expressions Options** menu command, you can select any amount of code you want. See the 'Expanding C++ statements' on page 203 for further information.
Examine class	Displays the selected class as the focus class in the Class Examiner.
Xref	Displays the selected class as the focus class in the Cross-Reference Browser.
Swap	Replaces a source file with the corresponding object file or replaces an object file with the corresponding source file.
Display	Displays in the Data Browser the variable or expression.
Display*	Displays in the Data Browser what the variable or expression points to.

As with the corresponding Workspace commands, the pop-up information windows only work if the selected item is currently in scope. For more information, see the entries for each of these commands in the *ObjectCenter Reference*.

Accelerators for Whatis and Print

ObjectCenter provides mouse accelerators to access the two pop-up information windows that you are most likely to use: **Print** and **Whatis**. You must have an expression selected (or the mouse pointer on an expression and nothing selected).

Print appears when you use the Shift key and the Left mouse button. **Whatis** appears when you use the Shift key and the Middle mouse button.

Table 24 contains ObjectCenter commands that give you information on your program, information that is available only in the Workspace.

Table 24 Workspace Commands for Code Visualization

Workspace Command	**Description**
dump	If the function is on the execution stack, shows all local variables (formals and automatics) for the selected function.
info	Lists the name, size, and type of the object associated with an address.
unres	Lists unresolved references.

For more information, see the **dump**, **info**, and **unres** entries in the *ObjectCenter Reference*.

Chapter 8 Customizing ObjectCenter

This chapter describes several ways you can tailor ObjectCenter to your own requirements. It covers the following topics:

- *Using the ObjectCenter startup files*
- *Using ObjectCenter options*
- *Customizing the Project Browser and integrating revision control systems*
- *Creating and managing customized buttons and menu items*
- *Connecting your editor to ObjectCenter*
- *Using Workspace commands with aliases*
- *Using C code to work with ObjectCenter*
- *Customizing key bindings*
- *Using eight-bit character sets*
- *Customizing the preprocessor for the load command*
- *Setting and examining environment variables*
- *Conditionalizing code in source files.*

Using ObjectCenter startup files

When you invoke ObjectCenter from the shell, ObjectCenter reads two startup files: a global initialization file named **ocenterinit** and a local initialization file named **.ocenterinit** (if you start in component debugging mode) or **.pdminit** (if you start in process debugging mode). These are text files in which you put any ObjectCenter commands, option settings, or C code that is accepted at the Workspace. ObjectCenter sends each line in the file to the Workspace and executes it.

You can customize how ObjectCenter begins each session by editing either your global or local ObjectCenter startup file.

Global startup file

Before reading your local startup file, ObjectCenter reads a global startup file, named **ocenterinit** (no initial period).

Typically, you use the **ocenterinit** file to set system-wide attributes such as:

- The directories ObjectCenter searches for libraries, header files, and so on
- The libraries that ObjectCenter attaches automatically
- ObjectCenter options and aliases for all users

NOTE The system-wide **ocenterinit** file is located in the directory **CenterLine/configs**.

See your system administrator if you do not know where the **CenterLine** directory is installed on your system.

Local startup file

After reading the global startup file, ObjectCenter looks for a local startup file in the current working directory; if it does not find the file there, it searches in your home directory. If you are starting ObjectCenter in component debugging mode, ObjectCenter uses the local startup file named **.ocenterinit**. For process debugging mode, ObjectCenter uses the startup file **.pdminit**. ObjectCenter does *not* read the global startup file, **ocenterinit**, in process debugging mode.

You use the local startup file to set the values of ObjectCenter options and define aliases that are used across ObjectCenter sessions. Here is a sample **.ocenterinit** file:

```
% cat .ocenterinit
/* Define aliases for common commands. */
alias s         step
alias n         next
/* Specify option settings. */
use . ../test ../src
setopt tab_stop 4
```

This startup file uses three ObjectCenter commands: **alias**, **use**, and **setopt**. As a result of ObjectCenter reading this file, the user can type **s** as an alias for **step** and **n** as an alias for **next**. Also, the directories that ObjectCenter searches when loading files are set, and the number of spaces for tab expansions is set to 4.

Because ObjectCenter first looks in the current working directory for **.ocenterinit**, you can have different **.ocenterinit** files for use with different projects, as long as you work in different directories.

The system-wide **ocenterinit** file is read before the local **.ocenterinit** file, so any specifications in the local file will override corresponding specifications in the system-wide file.

Restarting a session

You may want to restart an ObjectCenter session to clear your current session or to switch debugging modes.

Select **Restart Session...** on the **ObjectCenter** menu of the Main Window**.** The Restart Environment window appears enabling you to specify how you want to restart the session. The window has these fields:

- Runtime Engine

  Select **Component Debugger** to start a cdm session or **Process Debugger** to start a pdm session.

- Directory

  This is the directory in which you want ObjectCenter to look for local initialization files. The default is the directory in which ObjectCenter looked at last startup. Edit this field if you want to change the directory.

- Args

  In the Args field, enter the arguments with which you want ObjectCenter to restart.

- Load Global Initialization File

  Select **Load Global Initialization File** to initialize the new session using the global initialization file **ocenterinit**.

- Load Local Initialization File

  Select **Load Local Initialization File** to initialize the new session using the local cdm initialization file **.ocenterinit** or the local pdm initialization file **.pdminit**.

After you have entered the information about your restart, press the **Restart Debugger** button.

Using X resources

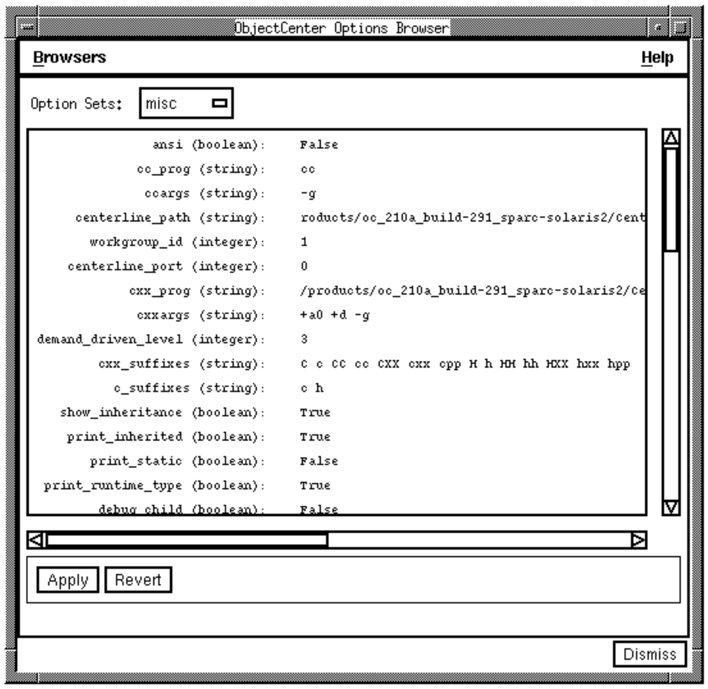
As with other X applications, you can customize the ObjectCenter GUI by specifying values for X11 resource variables. For information on X11 resources that you can modify to customize ObjectCenter, see the **X resources** entry in the *ObjectCenter Reference*.

Using ObjectCenter options

In component debugging mode, the Options Browser allows you to examine and change the values of all ObjectCenter options. You can customize much of ObjectCenter's functionality through these options. For information on each ObjectCenter option, see the **options** entry in the *ObjectCenter Reference***.**

Displaying options

To display the ObjectCenter options and their current values, from any primary window, display the **Browsers** menu and select **Options Browser**. Here is an illustration of the Options Browser:



Option Sets: misc

```
               ansi (boolean):    False
            cc_prog (string):     cc
             ccargs (string):     -g
     centerline_path (string):    roducts/oc_210a_build-291_sparc-solaris2/Cent
     workgroup_id (integer):      1
    centerline_port (integer):    0
           cxx_prog (string):     /products/oc_210a_build-291_sparc-solaris2/Ce
            cxxargs (string):     +a0 +d -g
demand_driven_level (integer):    3
       cxx_suffixes (string):     C c CC cc CXX cxx cpp H h HH hh HXX hxx hpp
         c_suffixes (string):     c h
  show_inheritance (boolean):     True
   print_inherited (boolean):     True
      print_static (boolean):     False
print_runtime_type (boolean):     True
       debug_child (boolean):     False
```

The **Options Sets** line shows the current option category, and the Options area lists each option grouped under that category. Each option is shown on a separate line. Each option line shows the name of the option, the type of value it takes (Boolean, integer, or string), and the current value setting.

Changing the option category

The **Options Sets** menu lists all of the categories under which options are grouped. To change the current option category, display the **Options Sets** menu and select the desired menu item.

Changing option settings

To change the value of an option, move the cursor to the value field for the desired option line, drag the mouse to select the portion of the field you want to change, and type the changes. The Options Browser has a limit of 1000 characters for each entry field. If you need to set an option to a value with more than 1000 characters, you must use the **setopt** command in the Workspace.

At the bottom of the screen, the Options Browser displays a brief description of the option you have in input focus.

Values that cannot be changed

A few options, such as the ObjectCenter version number, are uneditable. If you try to edit one of them, the Options Browser displays an error message at the bottom of the screen that says the option is read-only.

Canceling changes

At any point while making changes to option settings, you can cancel all your changes since you last applied new values. To revert to the option settings in effect when you last applied settings, select the **Revert** button.

NOTE When you revert, changes made since the last time you applied new values are canceled for all options in the Options Browser, not only those options currently shown in the Options area.

Applying changes

You can change as many value settings as you like before applying these settings. To apply the current values as new settings for all the options, select the **Apply** button.

You can apply changes to one field at time by pressing the Return key after you have entered a change in each field. If you apply a change with the Return key, however, ObjectCenter applies that value and the Revert key is unable to restore the previous value.

Changes made since the last time you applied new values are applied to all options in the Options Browser, not only those options currently shown in the Options area.

You can also enter the following in the Workspace:

```
-> setopt option_name
```

For more information on setting, unsetting, and viewing ObjectCenter options, see the **setopt**, **unsetopt**, and **printopt** entries in the *ObjectCenter Reference*.

Saving option settings

Changes you make to ObjectCenter options are *not* automatically saved across sessions. When you leave ObjectCenter, then return later, option values you changed are not remembered.

You can explicitly save option settings across ObjectCenter sessions in several ways, for example:

- Put these option settings in your local startup file **.ocenterinit**; see 'Using ObjectCenter startup files' on page 213.
- Save your current session as a project file and start your new session by loading this project file. The current settings for all options are automatically saved with a project file.
- Explicitly specify each option as part of a CL makefile target or a command file that you use to set up your project when you start a new session.

For more information on how to use these methods, see 'Loading components as a project' on page 68.

Customizing the Project Browser and integrating revision control systems

For information on customizing the Project Browser, particularly for integrating revision control systems, see the **X resources** entry in the *ObjectCenter Reference.*

Creating and managing customized buttons and menu items

In the Main Window, you can customize the GUI access to ObjectCenter functionality by creating, changing, or deleting any of the following graphical aids:

- Buttons for standard menu items
- Menu items for customized commands
- Buttons for your customized commands

Using buttons for standard menu items

You can create a button for any menu item on the menus in the menu bar of the Main Window. You can also change or delete an existing button. ObjectCenter stores information about customized buttons for standard menu items in the file **.octr_buttons**.

NOTE ObjectCenter automatically generates the **.octr_buttons** file and saves it in your home directory at the end of your ObjectCenter session.

Although the **.octr_buttons** file is an ASCII file, it is not intended for direct editing.

Adding a new button

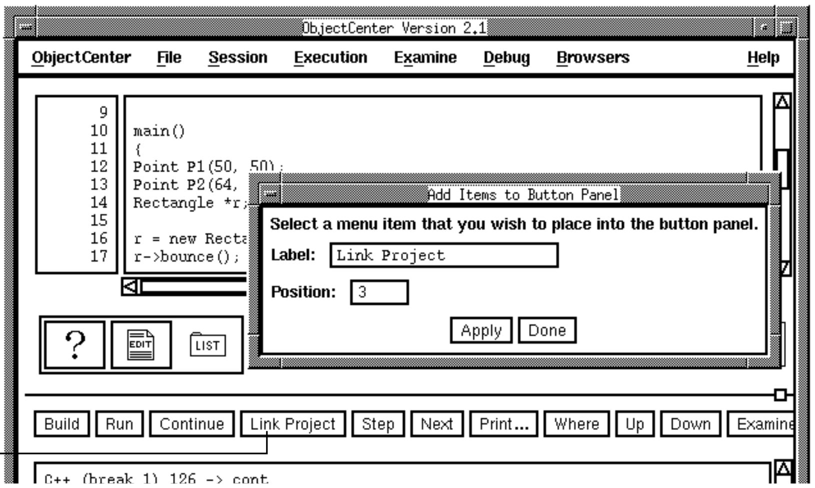
To add a new button, display the ObjectCenter menu and select **Button Panel**. In the submenu, select **Add Menu Items to Panel**. This opens the Add Menu Cell to Button Panel dialog box and places the Main Window in copy mode.

Display any of the menus in the Main Window menu bar and select the menu item for which you want to create a custom button. You cannot use a menu item that displays a submenu.

The name of the menu item appears on the Label line, and the Position line defaults to position 0 (the button at the far left). You can specify a new label and set the position for the button.

You use the **Apply** button in the dialog box to put the new button in the Button panel and get the GUI out of copy mode. The new button appears at the specified position on the Button panel.

For example, by displaying the **Session** menu and selecting **Link Project**, you could create a new **Link Project** button:



```
main()
{
Point P1(50, 50);
Point P2(64,
Rectangle *r;

r = new Recta
r->bounce();
```

Select a menu item that you wish to place into the button panel.

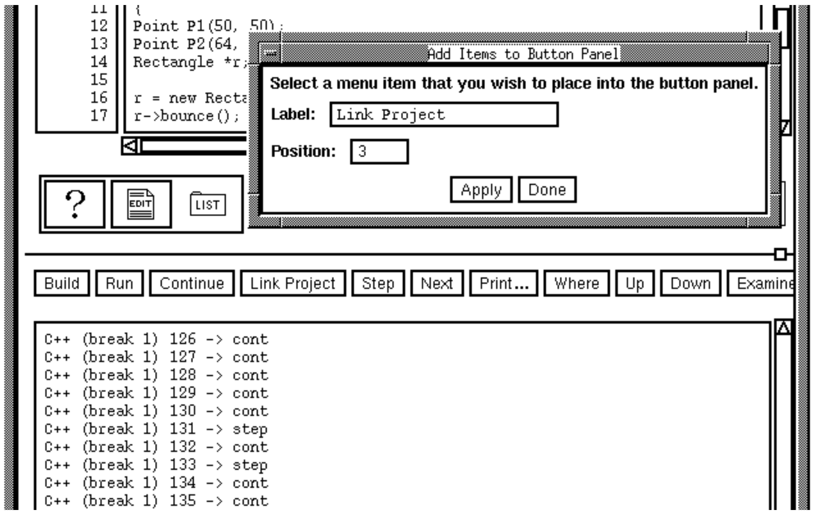
Label: Link Project

Position: 3

```
C++ (break 1) 126 -> cont
```

Changing or deleting buttons

To delete or to change the position or name of any button on the Button panel, display the ObjectCenter menu and select **Button Panel**. In the submenu, select **Customize Button Panel**. This opens the Modify Buttons in Button Panel window:



```
{
Point P1(50, 50);
Point P2(64,
Rectangle *r;

r = new Recta
r->bounce();
```

Select a menu item that you wish to place into the button panel.

Label: Link Project

Position: 3

```
C++ (break 1) 126 -> cont
C++ (break 1) 127 -> cont
C++ (break 1) 128 -> cont
C++ (break 1) 129 -> cont
C++ (break 1) 130 -> cont
C++ (break 1) 131 -> step
C++ (break 1) 132 -> cont
C++ (break 1) 133 -> step
C++ (break 1) 134 -> cont
C++ (break 1) 135 -> cont
```

The Buttons area shows all the buttons in the Button panel. To modify any of these buttons, select the button line in the Buttons area. You can specify a new name on the Button Label line or specify a new position on the Button Position line, then select the **Change** button. To delete the selected button, select the **Delete** button.

Using menu items and buttons for customized commands

You can create, change, or delete menu items or buttons for customized commands that you create. To create a customized command, display the ObjectCenter menu and select **User Defined**. In the submenu, select **Add, Change, Delete**. This opens the User Defined window. ObjectCenter stores information about customized buttons and menu items for customized commands in the file **.octrusrcmd**.

NOTE ObjectCenter automatically generates the **.octrusrcmd** file and saves it in your home directory at the end of your ObjectCenter session.

Although **.octrusrcmd** is an editable ASCII file, we recommend that you do not edit it. Rather, we recommend you use the **Add**, **Change**, and **Delete** choices in the GUI. If you do decide to edit it, ensure that each customized menu item or button has a unique, sequential user command number.

When you define a customized command, you can use the variables shown in Table 25:

Table 25 Variables for Customized Commands

Variable	Description
$pwd	ObjectCenter's current working directory.
$filename	The filename of the file in the Source area, relative to ObjectCenter's current working directory.
$filepath	The absolute filename of the file in the Source area.

Table 25 Variables for Customized Commands (Continued)

Variable	Description
$first_selected_char	The position of the first character selected on **$first_selected_line**. Character positions are numbered beginning with 1, and tabs are considered to be a single character. If no text is selected in the Source area, this keyword returns 0.
$first_selected_line	Starting line number of the Source area's current text selection. Lines are numbered beginning with 1. If no text is selected in the Source area, this keyword returns 0.
$last_selected_char	The position of the last character selected on **$last_selected_line.** Character positions are numbered beginning with 1, and tabs are considered to be a single character. If no text is selected in the Source area, this keyword returns 0.
$last_selected_line	Ending line number of the Source area's current text selection. Lines are numbered beginning with 1. If no text is selected in the Source area, this keyword returns 0.
$selection	The current contents of the X11 **PRIMARY** selection, interpreted as a string. If the current selection is not available or is empty, **$selection** is replaced with an empty string.
$clipboard	The current contents of the X11 **CLIPBOARD** selection. If the current selection is not available or is empty, **$clipboard** is replaced with an empty string.

TIP: Avoiding multiple-line selections for customized commands

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.

Creating a custom command for Workspace commands

To create a customized command as an alias for one or more Workspace commands, you type the new command name on the Label line. Select the **Workspace** button next to the Type label to specify the type of commands that will be in the definition. If you want this customized command to appear as a button in the control panel, select **Create Button** next to the Options label.

Then, in the Definition area, type the Workspace input that defines this custom command. 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. You can enter as many commands as you like. This allows you to create a batch of single-line Workspace commands that you invoke under one alias.

Creating a custom command for shell commands

To create a customized command as an alias for one or more shell commands, you type the new command name on the Label line. Select the **Shell** button next to the Type label to specify the type of commands that will be in the definition. If you want this customized command to appear as a button in the control panel, select **Create Button** next to the Options label.

For a customized command that is an alias for shell commands, you also specify the following items:

- The shell you want ObjectCenter to fork when you invoke this custom command.
- Whether you want ObjectCenter 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.

NOTE 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.

Then, in the Definition area, type in the shell commands that define this custom command. Put any input that the specified shell will accept. You can enter as many commands as you like. This allows you to create a batch of shell commands that you invoke under one alias.

Invoking a custom command

To invoke a custom command that you have created (composed of either Workspace or shell commands), if you have created a button for it, select the corresponding button in the second row of the Main Window control panel.

If you did not create a button for this command, display the ObjectCenter menu and select **User Defined**. In the submenu, select the customized menu item.

NOTE You cannot invoke customized commands from the Workspace.

Modifying a custom command

To change a custom command, select the command name in the Commands area of the User Defined Commands window and make any changes you want in the Specifications or Definition areas. Then select the **Change** button in the Commands area.

To delete a customized command, select the command name in the Commands area and select the **Delete** button.

Connecting your editor to ObjectCenter

ObjectCenter provides a great deal of flexibility in integrating your editor. The way this integration is implemented determines whether the connection between ObjectCenter and your editor is established from ObjectCenter or from your editor.

Connecting GNU Emacs to ObjectCenter

There are two ways to integrate ObjectCenter with GNU Emacs. You can connect your GNU Emacs session to ObjectCenter so that your Emacs session is used when you use the **edit** command or select an **Edit** symbol or button. You can also invoke ObjectCenter from within Emacs and use the Emacs main window. For more information, see the Emacs integration entry in the *ObjectCenter Reference*.

Integrating other editors

If you use an editor other than **vi** and **emacs** and want to integrate it into ObjectCenter's open architecture, look in the **CenterLine/API** directory for documentation and examples of writing edit servers with the CenterLine API.

Using Workspace commands with aliases

The **alias** command lets you establish additional identifiers for commands or other text. Aliases are often single-letter shortcuts for commonly used commands. The original command name remains valid.

To create an alias, use the **alias** command with this syntax:

```
-> alias alias_name command
```

For example, to create an alias **mylpr** that sends a file to the printer, you could issue this command:

```
25 -> alias mylpr sh lpr
```

With this alias defined, to print the file **main.c**, you could issue this command:

```
26 -> mylpr main.c
```

You can look at your current aliases by issuing the **alias** command without any arguments:

```
-> alias
ls                sh ls
pwd               sh pwd
assign            print
set               print
undisplay         sh echo "Use the ’delete’ command
to remove display items."
restore           sh echo "Use the ’load’ command to
restore project and image files."
mylpr             sh lpr
```

Aliases can take arguments. For information on specifying arguments with aliases, see the **alias** entry in the *ObjectCenter Reference*.

To learn how to save aliases across ObjectCenter sessions, see 'Using ObjectCenter startup files' on page 213.

Using C++ code to work with ObjectCenter

Using built-in CenterLine functions

To allow you to control debugging operations from ObjectCenter actions or from debugging statements within your source code, all ObjectCenter commands have a function equivalent that can be used to call the command from either C or C++ code.

How CenterLine commands work

The function equivalent for any command is the name of the command with the prefix **centerline_** added to it. For example, **centerline_stop()** causes an immediate break level in the same way that typing **stop** in the Workspace does.

All **centerline_*()** functions take one argument, a string. If the ObjectCenter command does not take an argument, you need to use an empty string as the argument when using the function equivalent.

The **centerline_*()** functions have the following prototype**:**

```
int centerline_command-name(char *);
```

For example, the prototype for the function to invoke the **stop** command is:

```
int centerline_stop(char *);
```

CenterLine functions without command equivalents

ObjectCenter also defines several C functions, described in Table 26, that do not have ObjectCenter command equivalents.

Table 26 Centerline Functions Without Command Equivalents

Function	Description
centerline_getopt()	Returns the value of an option
centerline_malloct()	Allocates memory with type checking
centerline_true()	Indicates whether ObjectCenter is running
centerline_unset()	Marks memory as having unset value
centerline_untype()	Marks memory as initialized and valid

Using CenterLine functions in actions

You can use CenterLine functions in actions to create conditional debugging procedures. For example, the following action stops execution, establishes a break level, and displays the current break and scope locations:

```
-> action on j
Enter body of action. Use braces when entering
multiple statements.
action -> {
action +> printf("j is %d\n", j);
action +> if ( j > 50 ) {
action +> centerline_stop("");
action +> centerline_whereami("");
action +> }
action +> }
action (2) set on address 0x1947e8.
```

Using CenterLine functions in your source code

You can also use CenterLine functions to place ObjectCenter debugging commands directly into your source code. Typically, you would conditionalize such code using the built-in ObjectCenter macros, such as **__**OBJECTCENTER**__**. For example:

```
draw(col_table[INDEX(count) - INDEX_DECREMENT],
     row_table[INDEX(count) - INDEX_DECREMENT]);
#ifdef __OBJECTCENTER__
centerline_stop("");
centerline_whereami("");
#endif
do_wait();
```

For information on ObjectCenter macros, see the **built-in macros** entry in the *ObjectCenter Reference*.

Using lint-style comments to suppress warnings

The comments in source code listed in Table 27 turn off ObjectCenter's load-time error checking for the specified violations:

Table 27 Lint-Style Comments to Suppress Load-Time Warnings

Comment	Action
/*VARARGS*/	Suppresses reporting that a function takes a variable number of arguments.
/*VARARGS*n**/	Suppresses reporting of a variable number of arguments, after *n* arguments.
/*NOTREACHED*/	Suppresses reporting that the following statement cannot be reached.
/*ARGSUSED*/	Suppresses reporting that formal parameters of a function are not used.
/*SUPPRESS *n**/	Suppresses reporting of violation #*n*. If the comment appears at the global level of a file, the violation is suppressed for the entire file. If the comment appears within a function, the violation is suppressed only for the following line.
/*EMPTY*/	Suppresses reporting on empty bodies, such as in *if* statements and *for* loops. The /*EMPTY*/ comment must appear on its own line preceding the statement on which reporting is to be suppressed.

For a complete list of all ObjectCenter warnings and error messages, see the **violations** entry in the online *Reference.*

Customizing key bindings

The command processor for the Workspace provides key bindings that support inline editing and input history. For example, you can move to the beginning of a line of input by pressing Control-a and scroll through previous lines of input by pressing Control-p.

Key sequences are bound to key functions or key commands:

- **Key functions** support cursor movement and inline editing
- **Key commands** treat the text of the binding as arguments that should be executed when they are called

You can display the list of key functions and commands by issuing **keybind** without arguments. The list is also available in the **keybind** entry in the *ObjectCenter Reference***.**

The example below binds the key Control-l to echo the string **load .c.** The Control-b characters move the cursor back before the suffix **.c**. The Control-v that you type is not echoed on the display; rather; it is used to prevent interpretation of the subsequent control character. The Control-l key sequence then expands to **load .c** with the cursor located before the **.c**.

```
-> keybind ^V^L macro load .c^V^B^V^B
-> ^L
-> load .c
```

For more information, see the **keybind** entry in the *ObjectCenter Reference*.

Using eight-bit character sets

ObjectCenter supports eight-bit character sets. Add the following two lines to your local **.ocenterinit** file so you can use the Meta key to get the extended character set:

```
setopt eight_bit
unsetopt line_meta
```

To turn on this feature for all users at your site, ask your system administrator to add the two lines to the global **ocenterinit** file.

Customizing the preprocessor for the load command

To filter source files through a special preprocessor before they are loaded with the **load** command, set the **preprocessor** option. The value of the **preprocessor** option should contain the argument **%s**, which is replaced with the name of the file being loaded.

For example, to pass all source files through the **m4** preprocessor before they are loaded, set the **preprocessor** option as follows:

```
-> setopt preprocessor m4 %s
```

To filter input to the Workspace through a preprocessor, you can bind the Return key to send all input to a subshell. For example, to send all input to the **m4** preprocessor, enter the following command (where RETURN is shown with **^M**):

```
-> keybind ^V^M user m4
```

Setting and examining environment variables

Environment variables make up an array of strings that are available to programs through the global variable **environ** and the formal parameter **envp,** which is passed as the third argument to **main()**. (For more information, see the UNIX documentation on **setenv** and **printenv**.)

ObjectCenter's **printenv**, **setenv**, and **unsetenv** commands manipulate environment variables within ObjectCenter. 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 ObjectCenter to control its own operation.

Table 28 Commands for Setting and Examining ObjectCenter Environment Variables

Command	Action
printenv	Displays the values of environment variables.
setenv	Sets the values of environment variables. If the second argument to **setenv** is omitted, the empty string ("") is used as the value.
unsetenv	Unsets environment variables.

For example, you would display and set the current setting for the **SHELL** environment variable in the following way:

```
-> printenv SHELL
SHELL=/usr/local/bin/tcsh
->
-> setenv SHELL /bin/sh
-> printenv SHELL
SHELL=/bin/sh
```

Note that **printenv** displays the default values of the environment variables, which are the values that the user's program inherits each time it starts. If a program alters an environment variable with the **putenv()** library function, the change is not shown by the **printenv** command.

Also, changing the **EDITOR** or **DISPLAY** shell variables with ObjectCenter's **setenv** command does not affect which editor, display screen, or paging program ObjectCenter uses. To modify ObjectCenter's behavior, use the Options Browser (see 'Using ObjectCenter options' on page 216).

Conditionalizing code in source files

You can load your source files into ObjectCenter without making any modification whatsoever. But there may be some debugging code that you want executed only when you are in ObjectCenter. When you compile the code, you do not want the ObjectCenter-specific debugging code included.

For your convenience, ObjectCenter predefines several macros, including __OBJECTCENTER__, that you can use to **#ifdef** your code so that certain code is used only when you are working in ObjectCenter. For example, your code would look like this:

```
< program code >

#ifdef __OBJECTCENTER__
< code to be run only when in ObjectCenter >
#endif

< more program code >
```

For more information, see the **built-in macros** entry in the *ObjectCenter Reference.*

Chapter 9 Using Ascii ObjectCenter

This chapter describes how working in Ascii ObjectCenter differs from working in the ObjectCenter GUI. It covers the following topics:

- *Introducing Ascii ObjectCenter*
- *Ascii ObjectCenter basics*
- *Managing your project*
- *Load-time violation checking*
- *Run-time violation checking*
- *Interactive debugging*
- *Suppressing linking messages*

Introducing Ascii ObjectCenter

Except where specifically indicated, this book presents ObjectCenter functionality as it appears in the graphical versions. This chapter presents the differences between using the graphical and the nongraphical user interfaces. The nongraphical version of ObjectCenter is called *Ascii ObjectCenter*.

Reasons for using Ascii ObjectCenter

You might choose to use Ascii ObjectCenter over the graphical versions for any of the following reasons:

- To run ObjectCenter on a nongraphical workstation.
- To gain quicker startup time or to reduce the amount of memory needed to run ObjectCenter.
- To debug GUI programs.

  Using Ascii ObjectCenter, you can debug from an ASCII terminal running alongside the X server. This allows you to more easily debug programs that grab mouse and keyboard I/O.
- To do automated test runs.

  Using I/O redirection with the **run** command and setting the **batch_load** option, you can automate program tests. For more information, see the **run** entry in the *ObjectCenter Reference* and 'Batch mode' on page 245.

Using the Workspace

Unlike the GUI versions, Ascii ObjectCenter has a single work area, the Workspace:

```
% objectcenter -ascii
ObjectCenter Version 2.1.1
Copyright (C) 1986-1995 by CenterLine Software,Inc.

For customer service call 1-617-498-3100,
or send email to
‘objectcenter-support@centerline.com’.

Attaching: /usr/lib/libc.so
Attaching: /usr/lib/libdl.so.1
Attaching: /.../lib/a0/libC.so
->
```

The Ascii ObjectCenter Workspace handles the same command and C++ and C code input and display of results as the Workspace in the GUI. However, in Ascii ObjectCenter, the Workspace is the total user interface. Therefore, the Workspace also takes on the Ascii-equivalent functionality of the Source area and all of the Browsers. In Ascii ObjectCenter, all of the available equivalent functionality of the GUI can be accessed through Workspace commands.

Accessing functionality

Since the Workspace is the only work area available in Ascii ObjectCenter, you use Workspace commands to access functionality equivalent to that offered in the GUI work areas.

Table 29 shows you what commands to use and gives you the context where this functionality is discussed for the GUI.

Table 29 Access to Functionality in Ascii ObjectCenter Compared with GUI Access

GUI Work Area	Equivalent Functionality in Ascii ObjectCenter	Where this Functionality is Discussed for the GUI
Source area in the Main Window	The **list** Workspace command lists lines of code that provide the context of the command argument.	'Listing source code' on page 43.
	The **action**, **stop**, **status**, and **delete** commands set, display, and remove debugging items.	'Using debugging items for interactive debugging' on page 115
Your editor	The **edit** command invokes your editor.	'Editing source code' on page 45.
Project Browser	The **load** command loads files individually.	'Loading individual components' on page 62.
	The **unload**, **swap**, **instrument**, and **uninstrument** commands manage files in your project individually.	'Managing individual components in your project' on page 75.
	The **build**, **link**, **unres**, **run**, and **rerun** commands manage your whole project.	'Managing your whole project' on page 82.

Table 29 Access to Functionality in Ascii ObjectCenter Compared with GUI Access

GUI Work Area	Equivalent Functionality in Ascii ObjectCenter	Where this Functionality is Discussed for the GUI
Properties window of the Project Browser	The **setopt** command sets the following project-wide options for loading: **ansi** **instrument_all** **load_flags** **path** **program_name** **swap_uses_path**	'Setting project-wide properties' on page 85.
Contents window of the Project Browser	The **contents** command lists the files and libraries in your current ObjectCenter project.	'Project Browser' on page 175.
Cross-Reference Browser	The **xref** command lists the references to and from the command argument.	'Cross-Reference Browser' on page 191.
Data Browser	The **display** command displays a summary of the data structure for the command argument.	'Data Browser' on page 197.
Error Browser	When load-time or run-time errors are encountered as you load files or run your program, violation messages appear in the Workspace. You manipulate these messages by using the **suppress** and **unsuppress** commands.	'Using the Error Browser to deal with warnings and errors' on page 101.
Options Browser	The **setopt** and **unsetopt** commands manipulate option settings.	'Using ObjectCenter options' on page 216.
Workspace	The **make**, **source**, and **load** *project_file* commands load files as a project.	'Loading components as a project' on page 68.

Ascii ObjectCenter basics

Starting Ascii ObjectCenter

To start Ascii ObjectCenter, use the following command at the shell:

```
% objectcenter -ascii
```

If you want to run process debugging mode only, use the following command at the shell:

```
% objectcenter -pdm -ascii
```

Running process debugging mode only means being without access to **clms** (CLIPC Message Server), the process that manages the interprocess communication among all Ascii ObjectCenter application services.

Switching between debugging modes

If you are using Ascii ObjectCenter and you wish to switch between component and process debugging modes, you must start a new session from outside the environment. Use the following ObjectCenter command:

```
-> quit force
```

Then at the shell, use the appropriate startup command to invoke Ascii ObjectCenter. For going from component debugging mode to process debugging mode, use the following command line:

```
% objectcenter -ascii -pdm
```

For going from process debugging mode to component debugging mode, use the following command line:

```
% objectcenter -ascii -cdm
```

Since component debugging mode is the default mode, you do not need to use the **-cdm** argument.

Editing in Ascii ObjectCenter

While you are using Ascii ObjectCenter, editing a file suspends ObjectCenter. You can return to ObjectCenter by suspending or quitting from the editor. If the editor was started from a line listing options following a load-time or run-time violation, the options line reappears when you suspend or quit from the editor.

Suspending an Ascii ObjectCenter session

If you are using Ascii ObjectCenter from the C shell (**csh** or **tcsh**), you can suspend it and return to the shell by pressing Control-z or by issuing ObjectCenter's **suspend** command. (You cannot do this if you started Ascii ObjectCenter from the Bourne shell (**sh**).) To return to Ascii ObjectCenter, type **fg**.

Quitting Ascii ObjectCenter

To quit Ascii ObjectCenter, use the following command:

```
-> quit force
```

If you want to save your current project as a project file, you can use the **save** command before you quit, or use the **quit** command. ObjectCenter will prompt you for a project file name.

Managing your project

To display the contents of your project in Ascii ObjectCenter, type **contents** at the workspace prompt:

```
-> contents
  object: centerline (C++)
  source: workspace (C++)
 library: /usr/lib/libc.so
 library: /usr/lib/libdl.so.1
 library: /dir/codecenter/arch/lib/a0/libC.so
  object: main1.o, debugging (C++)
  object: link.o, debugging (C++)
```

To display the definitions made in a file, issue the **contents** command and supply the name of the loaded file as an argument:

```
-> contents link.o
Contents of   object: link.o, debugging (C++)
/my_dir/c++tutor_dir/link.C
/usr/include/stdio.h
/my_dir/c++tutor_dir/link.C
/my_dir/c++tutor_dir/link.h
/my_dir/c++tutor_dir/link.C
/my_dir/c++tutor_dir/link.h
/usr/include/stdio.h
/usr/include/sys/feature_tests.h
typedef int (*)() ;
struct  {...} ;
typedef unsigned int size_t ;
typedef long fpos_t ;
class FILE {...} ;
typedef class FILE FILE ;
class Link {...} ;
Link::Link(class Link *)
class Link *Link::nextLink()
char Link::setNext(class Link *)
```

Load-time violation checking

When you load a file in Ascii ObjectCenter, each load-time violation is reported immediately, and Ascii ObjectCenter prompts you to act. Here is a sample load-time warning message:

```
-> load shapes.C
Loading (C++):

----------------
““shapes.C”:72, `;’ missing after statement (Error
#573)
    71:   filled = 0
 *  72: }
    73:
`;’ missing after statement.
Unloading: shapes.C
Warning: 1 module currently not loaded.
Options: quit/reload/edit/abort [q] ?
```

The first line of the message displays the location of the violation followed by a brief description and violation number. The next three lines list the source code at the location of the violation, with the line containing the violation marked by an asterisk (*). This is followed by a description of the violation.

The last lines of the message contain a set of options from which you can select a specific action. The default option is enclosed in square brackets ([ ]). You can select the default option by pressing the spacebar or Return key.

The options available to you depend on the context and severity of the violation.

Handling warnings

See Table 30 for a list of ways to deal with a load-time warning.

Table 30 Handling Load-Time Warnings in Ascii ObjectCenter

Choice	Action
c	(Default). Continues loading the file.
s	Silences reporting of all warnings for the current file and all other files specified on the command line.
q	Quits loading the file.
a	Quits loading the file and aborts loading for other files specified on the command line.
e	Edits the file at the location of the violation.
r	Reloads the file if modified.
E	Suppresses the warning everywhere.
F	Suppresses the warning in the current file.
L	Suppresses the warning on the current line.
P	Suppresses the warning in the current procedure.
N	Suppresses the warning for the current name.

Handling errors

You have fewer choices when dealing with a load-time error; see Table 31 for a list of ways to deal with load-time errors.

Table 31 Handling Load-Time Errors in Ascii ObjectCenter

Choice	Action
c	(Default). Continues loading the file.
q	Quits loading the file.
r	Reloads the file if modified.
e	Edits the file at the location of the violation.
a	Quits loading the file and aborts loading for other files specified on the command line.

Listing source code

In Ascii ObjectCenter, the source listing appears immediately after the **list** statement. Ascii ObjectCenter displays the number of lines specified by the **page_list** option. If more lines can be displayed, **list** displays a **more** prompt, which accepts many of the options accepted by the shell's **more** utility. Type **h** to see the list of commands that can be entered.

Batch mode

You can set the **batch_load** option if you do not want to be prompted at each reported warning or error. Ascii ObjectCenter will display the messages, but it will continue loading the file automatically without prompting you.

See the **options** entry in the *ObjectCenter Reference* for more information about **batch_load**.

Run-time violation checking

In Ascii ObjectCenter, when a run-time violation occurs, you see a display like this:

```
-> run
Executing: Bounce

----------------
“main1.C”:12, main(),  (Error #156)
    11: {
 *  12: Point P1(50, 50);
    13: Point P2(64, 20);
Calling undefined function Point::Point(int,int).
Options: break/quit/edit/reload [b] ?
```

The first line of the message displays the location of the violation followed by the violation number. The next three lines list the source code at the location of the violation, with the line containing the violation marked by an asterisk (*). This is followed by a description of the violation.

The last lines of the message contain a set of options from which you can select a specific action. The default option is enclosed in square brackets ([ ]). You can select the default option by pressing the Spacebar or Return key.

The options available to you depend on the context and severity of the violation; see Table 32 for a list of options along with ways to handle each.

Table 32 Handling Run-Time Violations in Ascii ObjectCenter

Option	Result
b	(Default). Generates a break level.
c	Continues execution.
q	Quits execution.
e	Edits the file at the location of the violation.
r	Reloads the file if modified.
E	Suppresses the warning everywhere.
F	Suppresses the warning in the current file.
L	Suppresses the warning on the current line.
P	Suppresses the warning in the current procedure.
N	Suppresses the warning for the current name.

Handling spurious used-before-set messages

Since ObjectCenter has no direct knowledge of the operations executed within object code, run-time violations are difficult to detect when execution moves between source and object code. This is particularly true of dynamic used-before-set violations, since memory can be initialized within object code.

To handle this situation, ObjectCenter stores the value of the **unset_value** option (by default **191**) in each byte of uninitialized data. (This is not true of global and static variables that lack explicit initializers; these variables are initialized to 0, as the C++ language requires.) The assumption is that, if the data is initialized in object code, the data stored will not have the value **191** stored in any byte.

This assumption sometimes fails when data is read into memory with an object code library function, such as **read()** or **fread()**. This may cause spurious used-before-set warnings if the value stored equals **191**.

There are several ways to get rid of these warnings:

- Prevent them by inserting calls to **centerline_untype()**, a built-in ObjectCenter function that marks memory as initialized and valid. The **centerline_untype()** function is similar to the ObjectCenter **touch** command, except that it is easier to use in programs and will not mark unknown memory.

  For an example of using **centerline_untype()**, see the description of **centerline_untype()** in the *ObjectCenter Reference***.**
- Suppress the warnings.
- Change the default value of **191** by using the command **setopt unset_value**. In particular, **setopt unset_value 0** prevents further dynamic used-before-set warnings for the entire program; it effectively disables checking for used-before-set violations.

Interactive debugging

You use the following ObjectCenter commands for interactive debugging from the Workspace:

Table 33 Commands for Interactive Debugging in Ascii ObjectCenter

Workspace Command	Description
action	Specifies statements to execute when execution triggers the action. Allows you to customize conditional breakpoints.
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.
expand	Lists the functions that could be called from a C++ statement.
file	Displays and sets the current list location.
info	Displays the name, size, and type of the item associated with an address.
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.
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.

Setting breakpoints at a line of code or on a function

In Ascii ObjectCenter, you use the **stop at** command to set a breakpoint on a line of code. You give the file and line number as arguments to the command. For example, to set a breakpoint at line 11 in the file **main1.C**, you would enter:

```
-> stop at “main1.C”:11
stop (1) set at “main1.C”:11, foo(void).
->
```

You can set a breakpoint in a function, which causes ObjectCenter to stop execution at the first line of the function. You do this using the **stop in** construction. For example, to set a breakpoint in the function **main()**, you would enter:

```
-> stop in main
stop (2) set at “main1.C”:23, main().
->
```

When you list your code, ObjectCenter uses a **B** to indicate a line with breakpoints set.

Setting breakpoints in library functions

You can set a breakpoint in a library function only if the function has been linked in. If you want to set a breakpoint on a library function before running your program, you can use the **link** command to explicitly link unresolved symbols from static libraries. Then set the breakpoint.

Here is an example of setting a breakpoint on a library function without first running the program.

```
-> load bpprog.C
Loading (C++): bpprog.C
-> stop in printf
Cannot set stop or action on an undefined symbol:
’printf’.
-> link
-> stop in printf
stop (1) set at “/lib/libc.sl”, function printf().
->
```

Setting breakpoints on addresses

You can set a breakpoint on an address, global variable, or lvalue, which interrupts execution whenever a specific address is modified. You can set these breakpoints on the addresses of global variables, allocated data, formal parameters, and automatic variables. (The variables must be in scope when you set the watchpoints.)

If the watched address is modified within code loaded in object form, ObjectCenter does not detect the event and execution of the program is not interrupted. That is, this type of breakpoint is only triggered when the address is modified by code loaded in source form. To avoid spurious messages, the watchpoint does not trigger if the watched address is modified by you in the Workspace.

You use **stop on** to set a breakpoint on an address. The following example sets a breakpoint on the variable **abc:**

```
-> int abc;
-> stop on abc
stop (4) set on address 0x194788.
```

The number of bytes watched equals the size of the type of data. In the example above, four bytes are watched because **abc** is an **int**, and the size of an **int** is four bytes.

To set a breakpoint on the address stored in a pointer, the argument to **stop** should be the value of the pointer, as shown:

```
-> int *ptr;
-> ptr = (int*) malloc(sizeof(int[20]));
(int *) 0x40129f30 /* (<data>) (allocated) */
-> stop on *ptr
stop (1) set on address 0x40129f30.
->
```

ObjectCenter uses a **B** to indicate lines with breakpoints set.

Setting actions

You can tell ObjectCenter to execute certain statements whenever it reaches a particular location in your program by setting **actions**. You can choose whether ObjectCenter generates a break level as part of the action.

You use the Workspace to specify the statements to execute, so the easiest way to set an action is using the **action** command in the Workspace.

The **action** command takes the same arguments that **stop** does, so you specify the location of the action (for example, a line of code, a function, an address) the same way as with **stop**.

If you issue **action** without providing a location argument, the debugging action is executed at every line in your program. This allows you to set actions that constantly monitor the execution of your program. (In order to avoid spurious messages and infinite recursion, statements that are executed directly in the Workspace or by other debugging commands are not monitored.)

After you issue the **action** command, you are prompted to enter the **body of the action**.

The body of an action can consist of one or more C++ statements; enclose multiple statements within braces. (If you are debugging C code in C mode, actions consist of C statements.) Here is an example of setting an action.

```
-> action at “sample.C”:8
Setting action at “sample.C”:8, main().
Enter body of action. Use braces when entering
multiple statements.
action -> printf(“print %d”, i);
action (1) set.
```

ObjectCenter uses an **A** to indicate lines with actions set.

Specifying conditional actions

You can also conditionalize actions. For example, you can tell ObjectCenter to generate a break level only if certain conditions are true. The following example prints “i=3” if **i** = 3.

```
-> action at “sample.C”:14
Setting action at “sample.C”:14, main().
Enter body of action. Use braces when entering
multiple statements.
action -> if (i==3) printf(“i=3”);
action (1) set.
-> action at “sample.C”:14
Setting action at “sample.C”:14, main().
Enter body of action. Use braces when entering
multiple statements.
action -> if (i > 3) centerline_stop (“”);
action (2) set.
```

In addition to printing “i=3” in the program’s output window each time execution reaches line 14, the action interrupts execution with **centerline_stop()** if **i** is greater than 3.

For example:

```
-> run
Executing: a.out

Hello 0
Hello 1
Hello 2
Hello 3 i=3
Hello 4 Stopped in action #2, line 1, set in
main() at “sample.C”:14
13: printf(“\nHello %d “, i);
A 1: if (i==3) printf(“i=3”);
A* 1: if (i > 3) centerline_stop (““);
14: }
15: bye();
(break 1)->
```

Since ObjectCenter’s **stop** command is being called in code (instead of being typed in the Workspace), the function version of the **stop** command, **centerline_stop()**, is used. To learn more about calling ObjectCenter commands in code, see 'Using C++ code to work with ObjectCenter' on page 228.

Tracing program execution

You can use the **trace** command to trace through your code as it executes. You can only trace in source code. This example shows how to trace each line of your code:

```
-> load sample.C
Loading (C++): sample.C
-> trace
trace (1) everywhere.
```

This example shows how to trace in a function:

```
-> trace bye()
trace (1) set on function bye(void).
->
```

Tracing is a good way to follow the path your program takes.

Statements executed within an action are not traced. You can only trace through code loaded in source form. If you turn tracing on in Ascii ObjectCenter, all lines that are executed are echoed to the Workspace.

Limiting extent oftracing

You can issue **trace** with the name of a function to limit tracing to that function. (This feature is available only for functions loaded in source form.)

Turning tracing off

You turn off tracing by deleting the corresponding debugging item, as described in the next section.

Examining and deleting items

At any point, you can see which debugging items are set, and you can delete particular items.

Examining debugging status

To examine the debugging items currently set, issue the **status** command. The **status** command lists all debugging actions by number.

Deleting debugging items

From the Workspace you can issue the command **delete** *number*, where *number* is the number of the debugging item shown by the **status** command.

If you issue **delete** without specifying an argument, ObjectCenter deletes all debugging items at the current break location.

If you issue **delete all**, ObjectCenter deletes all debugging items everywhere.

Working in object code

You can set breakpoints and define actions in code loaded in source form as well as in code loaded in object form if the code was compiled with the **-g** option (these files contain debugging information).

In object code loaded without debugging information, you can stop on or set an action on a function name, but not on a particular line of code. You cannot set breakpoints or actions on an address, lvalue, or variable in object code, and you cannot set tracepoints on or trace through code loaded in object form.

More information

For more information, see 'Interactive debugging from Workspace break levels' on page 126.

Suppressing linking messages

By default, ObjectCenter displays a message when linking from a library:

```
Linking from ... Linking completed.
```

You can suppress the linking messages by setting the environment variable **CENTERLINE_LINK_SILENT** before starting ObjectCenter. This is particularly useful in Ascii ObjectCenter when you are linking from shared libraries; run-time linking messages will not obscure your program's output.

Index

This index covers the ObjectCenter User's Guide (page numbers prefaced with U) and the ObjectCenter Reference (page numbers prefaced with R).

Index