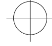
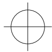
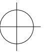


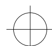
ObjectCenter Tutorial

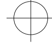
Version 2.1.1





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





CenterLine Software, Inc. reserves the right to make changes in specifications and other information contained in this publication without prior notice. The reader should in all cases consult CenterLine to determine whether any such changes have been made.

This Manual contains proprietary information that is the sole property of CenterLine. This Manual is furnished to authorized users of ObjectCenter solely to facilitate the use of ObjectCenter as specified in written agreements.

No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means without prior explicit written permission from CenterLine Software.

The software programs described in this document are copyrighted and are confidential information and proprietary products of CenterLine Software.

CenterLine and ViewCenter are registered trademarks of CenterLine Software, Inc. CodeCenter, ObjectCenter, ResourceCenter, and TestCenter are trademarks of CenterLine Software, Inc.

All other products or services mentioned in this document are trademarks or registered trademarks of their respective holders.

Licensed under one or more of U.S. Pat. Nos. 5,193,180 and 5,335,344; other U.S. and foreign patents pending.

© 1986 – 1995 CenterLine Software, Inc.
All rights reserved.
Printed in the United States of America.

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

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

ftp.centerline.com

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

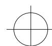
/pub/TOOLS/PDM.TAR.Z

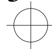
The following file on that host contains the source for the CenterLine C Preprocessor:

/pub/TOOLS/CLPP.TAR.Z

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

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





Using this book

ObjectCenter is an interactive C and C++ programming environment that provides two debugging modes: process debugging mode and component debugging mode. Component debugging mode includes an interactive C++ Workspace. This guide introduces you to ObjectCenter so that you can start using it to enhance or debug your code.

Read this preface before you begin the tutorial for an overview of the following topics:

- Using this guide
- Starting ObjectCenter and using its windows
- Finding information online
- Using the tutorial
- Setting up the tutorial directory

What this guide is about

This guide contains a tutorial that introduces you to the ObjectCenter environment. It also shows you how to load your own code into ObjectCenter, how to debug, test, and prototype your code, and how to set up and customize your environment.

Part I is a tutorial that takes you on a tour of the environment while you complete several debugging and code development tasks.

Part II shows you how to get started with your own code.

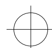
We recommend that you read through this introduction for a brief overview, and then go on to page 3 to begin the tutorial. We've tried to make the tutorial work the same way on all platforms. However, there may be differences on some platforms. Please read your *Release Bulletin* to see if there are differences before beginning the tutorial.

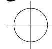
If you prefer to start by loading in your own code, go to page 63 after reading this introduction.

What you should know before starting

This guide doesn't assume any knowledge of ObjectCenter, but it does assume that

- You are familiar with the C++ language. It does not attempt to teach C++ programming.
- You are familiar with UNIX®, the X Window System™, and either the OPEN LOOK® or Motif® Graphical User Interface.





Conventions used in this guide

This guide uses the following conventions:

- To *display a pull-down menu*, move the mouse pointer over the menu title and press either the Left mouse button (Motif GUI) or Right mouse button (OPEN LOOK GUI).
- To *select a menu item*, hold down the same mouse button, drag the mouse pointer to the specified menu item, and then release the mouse button.
- To *select a button*, move the mouse pointer over the button and click the Left mouse button.

Starting ObjectCenter

Before you start using ObjectCenter, choose the editor ObjectCenter invokes when you select an edit symbol or enter the **edit** command, and set the DISPLAY variable to the system you will be using. 'Setting up your environment' on page 99 describes some other options and customizations you may want to set up when you are more familiar with ObjectCenter.

Specifying your editor

ObjectCenter supports **vi** and FSF GNU Emacs. The default editor is **vi**. To specify GNU Emacs as your editor, use the following shell command:

```
% setenv EDITOR emacs
```

GNU Emacs and **vi** are the *only* editors we support. You may be able to connect other editors to ObjectCenter. See 'Using your own editor' on page 107 for more information.

You can also start ObjectCenter under the control of FSF GNU Emacs. To see how, read the emacs integration entry in the *ObjectCenter Reference.*

Setting your display

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

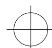
```
% setenv DISPLAY baxter:0
```

Invoking ObjectCenter

You invoke ObjectCenter with the **objectcenter** command.

```
% objectcenter
```

By default, ObjectCenter starts in component debugging mode with the default graphical user interface style for your platform. The illustrations in this guide show the Motif user interface.



The Main Window

When you invoke ObjectCenter, you see the Main Window, which acts as the hub of the ObjectCenter environment. The Main Window contains the Source area, which displays your source code, the Workspace, in which you enter commands and prototype code, and buttons and menus, for fast access to commands and browsers. ObjectCenter's graphical browsers open automatically when you enter the appropriate commands or make menu selections. You can also open any browser or raise it in front of other ObjectCenter windows from the Browsers menu.

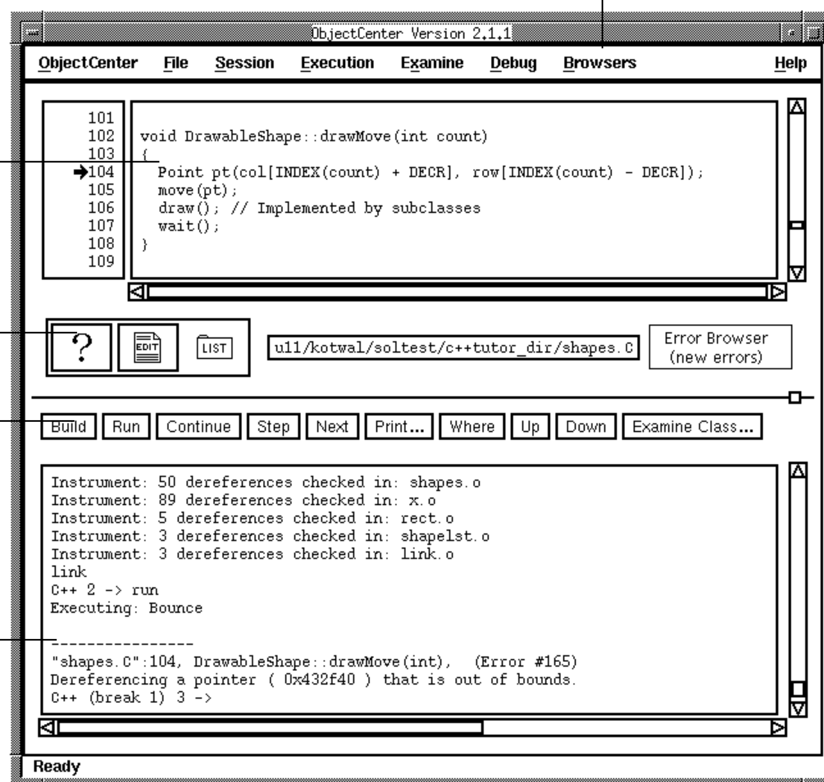
Browsers menu

Source area

? button for online documentation

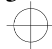
Button Panel

Workspace



Managing windows

If you have finished with a window you have opened, you can select the **Dismiss** button on any primary window or the **Cancel** button on other windows to close the window. You can also use your window manager's controls to iconify Browsers. Resize a window or use the horizontal and vertical scroll bars to view any graphical code representation too large to fit completely in the window.



Finding information online

The ObjectCenter product comes with a complete set of online manuals including a *User Guide* and *Reference*. There are also several other ways to access information online.

Manual Browser and "?" button

The Manual Browser contains the full text of the *ObjectCenter User Guide* and *ObjectCenter Reference*, information about new features and platform-specific differences, and answers to frequently asked questions. You can display the Manual Browser using three different methods:

- From any primary window in the debugger, display the **Browsers** menu and select **Manual Browser**.
- Click on the "?" button in the Main Window.
- Issue the **cldoc** command from a shell.

In the left panel of the Library window are one or more collections of books. Click on the name of a collection to display the names of books in that collection in the Books panel. Open a specific book by double-clicking on its name, or by selecting its name and clicking the Open button.

The man command

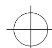
You can also invoke the *ObjectCenter Reference* by typing **man** in the ObjectCenter Workspace followed by the name of a command. For example

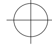
```
-> man debug
```

Searching the online documentation

You can perform searches from the Library window or from a book window. In the book window we've provided several search forms to simplify searching. Select **Forms** from the **Search menu** to see the list of forms. To expand or constrain your search, you can also use wildcards or Boolean expressions in the standard search form. For example:

- To locate all instances of the word profiling, enter **profiling** in the search field.
- To locate words such as profile and profiles as well, enter **profil*** in the search field.
- To locate all instances of the word profiling that appear in proximity to the word library, enter **profiling and library** in the search field.
- To locate all instances of the phrase "run-time error checking", enter **run-time error checking** in the search field.





The *Dynatext Reader Guide* in the doc_info collection describes more advanced search techniques.

Moving from place to place

Use the Next, Previous, and Go Back buttons to navigate through the book. Underlined text is hot—clicking on it scrolls the window to the section of the book referenced, or opens a new window if the reference is to another book. You can create a history of your movement through the book by selecting New Journal from the File menu and selecting Start Record in the dialog that pops up.

Printing and copying

Print sections by selecting Print from the File menu and highlighting the sections you want to print. Export sections to a file by selecting Export from the Edit menu, highlighting the sections you want to export, selecting Content as the Export format, providing a filename, and clicking the Export button.

Context-sensitive help

You can get context-sensitive online help by placing your cursor over any item in the graphical user interface and pressing the F1 or Help key, or by selecting "On Context" from the Help menu and placing the "?" cursor over an item in the interface.

Using the Help menu

In addition to context-sensitive help, the ObjectCenter GUI also offers help on a range of topics. You access this help through the **Help** menu in the Main Window or any of the graphical browsers.

Workspace help

Get quick help on commands and options by entering **help** followed by a command or option name in the Workspace. For example, to get a usage summary for the **email** command:

```
-> help email
```

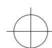
CenterLine manual pages

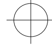
Access CenterLine manual pages for shell commands and library functions from a shell with the **man** command, or from the Workspace with **sh man**, for example

```
-> sh man cldoc
```

You may need to set your MANPATH environment variable before starting ObjectCenter. For example:

```
% setenv MANPATH path/CenterLine/man:$MANPATH
```



How to use the tutorial

To explore the ObjectCenter programming environment, the tutorial uses a simple program named Bounce. The Bounce program creates a new window and bounces a rectangle within the window. In the first three tasks in the tutorial, you will find an error in the program, and then enhance it so that it bounces a linked list of circles.

The tutorial provides a series of tasks for exploring ObjectCenter features:

- **Correcting run-time errors.** You learn how to set up a project, use load-time and run-time error checking to track down errors, and edit and reload modules.
- **Enhancing a program interactively.** You learn how to examine class hierarchies, examine the calling structure of a program, and prototype enhancements in the Workspace.
- **Adding C++ template code to a program.** You learn how to create objects from templates using the ObjectCenter automatic instantiation process.
- **Loading the Bounce program with the load command.** You learn how to load a program into ObjectCenter without a makefile.
- **Speeding up compilation with precompiled header files.** You learn how to establish rules for skipping header files and thus avoid redundant recompilation.

The first three tasks are designed to be performed in sequence. First you use the ObjectCenter **make** command to set up an application that bounces a rectangle and then you correct run-time errors in the program. The second and third tasks build on the first, enhancing and adding template code to the Bounce program.

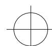
The last two tasks can be completed independently. They are designed to give you some experience in using the **load** command to set up your application, and in using the precompiled header file facility.

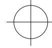
Setting up the tutorial directory

To set up the tutorial directory, go to your home directory and invoke the **c++tutor** command, as follows:

```
% cd
% c++tutor
```

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

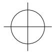


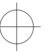


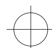
The **c++tutor** command creates a directory called **c++tutor_dir** in the current directory and copies the tutorial files to the new directory. The **c++tutor_dir** contains the following files:

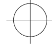
List.c	**appVector.C**	**rect.C**	**table.c**
List.h	**circle.C**	**rect.h**	**table.h**
Makefile	**circle.h**	**shapelst.C**	**x.C**
String.C	**link.C**	**shapelst.h**	**x.h**
String.h	**link.h**	**shapes.C.orig**	**x_image.h**
Vector.c	**main1.C.orig**	**shapes.h**	
Vector.h	**main2.C**	**skip**	

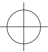
Most of these files are used in the tutorial. A few (for example **appVector.C**, **Vector.c**, and **Vector.h**) are used elsewhere in the documentation.

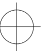


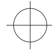












Contents

Using this book iii

Part I Tutorial 1

Chapter 1 Correcting run-time errors in the Bounce program 3

Getting started 5
Setting up the Bounce project 5
Viewing the project components 7
Running the Bounce program 9
Swapping a file from object to source form 11
Exploring the code to understand the problem 12
Fixing the error and rebuilding your program 15

Chapter 2 Enhancing the program 17

Examining classes with the Inheritance Browser 19
Adding a class to the program 21
Using the Class Examiner to view individual class members 22
Looking at the program's structure in the Cross-Reference Browser 24
Prototyping your enhancements 26
Modifying the program 30

Chapter 3 Adding template code to the program 33

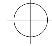
Starting with Chapter 3 35
Instantiating a class template in the Workspace 36
Using the **expand** command 39
Loading code that uses templates 40
Examining your code with the Data Browser 41

Chapter 4 Using the load command 45

Starting Chapter 4 47
Setting options 48
Loading the Bounce program 49
Linking from libraries 51

Chapter 5 Speeding up compilation with precompiled headers 53

Creating the skip information file 55
Recompiling with header-file skipping 56
A more complex skip information file 57



Part II Getting started with your own code 59

Chapter 6 Loading your application 61

Getting ready to load your code 63
Loading your application with the load command 65
Loading your application with CenterLine makefile targets 66
Creating a makefile with the EZSTART utility 70
Saving your project 71
Troubleshooting the load and link commands 73

Chapter 7 Running your application 75

Running your program 77
What is run-time error checking? 78
Adding more run-time error checking to object files 79
Responding to run-time problems 80
Swapping a file from object to source 82
Debugging techniques 84
Rebuilding your project 92
Exploring, enhancing, and testing your application 93
Troubleshooting run-time issues 97

Chapter 8 Setting up your environment 99

Setting options in the Workspace 101
Saving your option settings 104
Customizing your environment 105

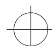
Index 109

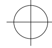
List of Tables

Table 1 Resolving **load** and **link** Problems 73
Table 2 Types of Run-Time Error Checking 78
Table 3 Troubleshooting Swapping 83
Table 4 Troubleshooting Run-Time Issues 97
Table 5 Additional Workspace Options 103

List of Tips

Saving load-time error messages to a file 74
Exporting the contents of the Project Browser to a text file 98

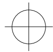


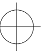


Part I: Tutorial

This part of the manual contains a tutorial that takes you on a tour of the ObjectCenter environment. The tutorial has five chapters.

In the first three chapters you set up, debug, and enhance a program that bounces shapes. These chapters are designed to be completed in sequence:

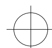


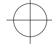


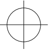
- *Correcting run-time errors in the Bounce program*
- *Enhancing the program*
- *Adding template code to the program*

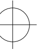
Work through the last two chapters to learn how to

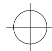
- *Use the load command to get code into ObjectCenter*
- *Speed up compilation with precompiled headers*

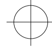






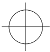
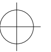






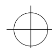
Chapter 1 Correcting run-time errors in the Bounce program

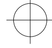
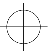
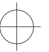
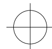
For the purposes of this tutorial, the Bounce program represents code that you have inherited and plan to enhance. The program creates a new window and bounces a rectangle in the window. The existing code has a problem that you will find and fix.

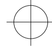



To begin, you load the Bounce program in component debugging mode, establish it as a project, and run it. In doing so, you learn about the following aspects of using ObjectCenter:

- *Setting up a project*
- *Running a program in component debugging mode*
- *Finding a problem using run-time error checking*
- *Swapping a file from object to source form*
- *Exploring the code to understand the problem*
- *Fixing the error*
- *Rebuilding and running the corrected program*





Getting started

To begin the tutorial:

1 If you haven't already created the tutorial directory, install the examples as described in 'Setting up the tutorial directory' on page viii.

2 Go to the tutorial directory and invoke ObjectCenter:

```
% cd ~/c++tutor_dir
% objectcenter
```

Setting up the Bounce project

When you establish a project in component debugging mode, you load all the source, object, and library files related to the application on which you are working.

When starting, ObjectCenter automatically loads the standard C library and the standard C++ library. On some platforms, ObjectCenter loads shared versions of these libraries.

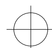
NOTE See your system administrator if ObjectCenter was unable to find either the C or C++ library when it started up.

Using a makefile to set up a project

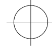
You can load files individually into ObjectCenter, or add special targets to your makefile that ObjectCenter can use to load files. For this tutorial, we have prepared a makefile target that loads all the files in the Bounce project. To invoke it, enter the following Workspace command:

```
C++ 1 -> make bounce_project
```

The ObjectCenter **make** command uses special makefile rules designed to work with ObjectCenter. The rules in the **bounce_project** target load in the separate files that constitute the components of the project for the Bounce program.



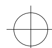
evaloc.book : octutor1 6 Tue Jun 6 12:24:32 1995



ObjectCenter displays messages in the Workspace as it loads, makes, and links the appropriate files.

NOTE The output of commands may vary from platform to platform. The illustrations in this guide show the output on the Solaris™ 2.3 platform. There may be other variations in behavior from platform to platform. For more information, please refer to your *Release Bulletin* and the "anomalies" section in the *Platform Guide* appendix for your platform in the online *Reference*.

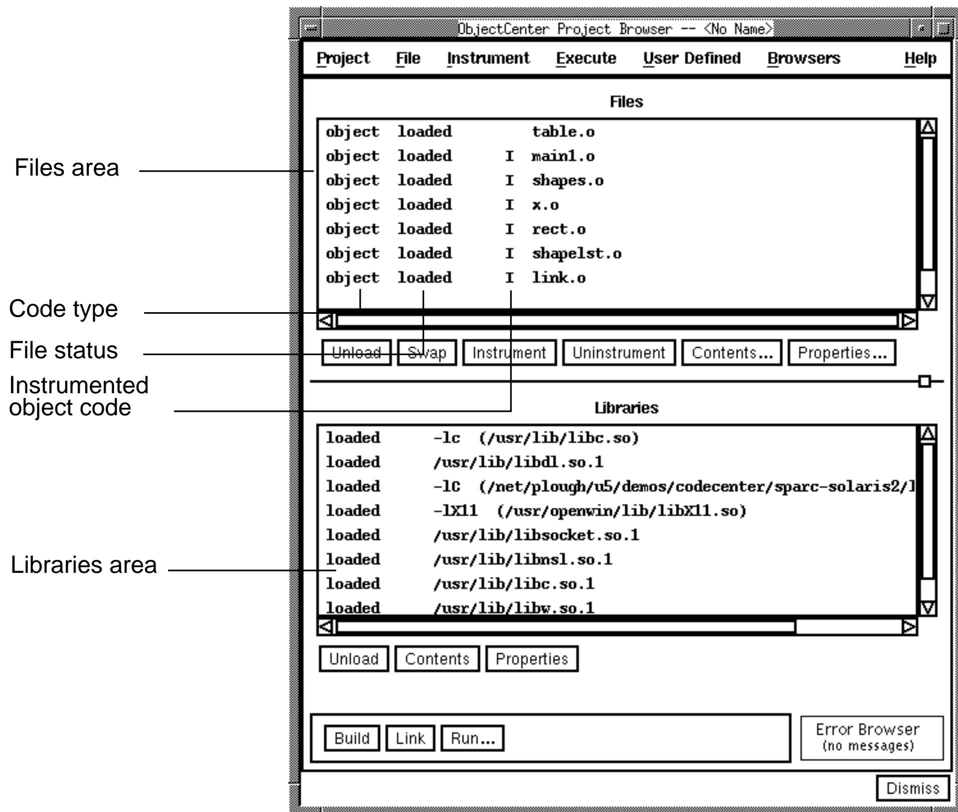
If you didn't have a makefile, you could load your application using the **load** command. We show you how to load the tutorial files with the **load** command in 'Using the load command' on page 55.



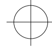
evaloc.book : octutor1 7 Tue Jun 6 12:24:32 1995

Viewing the project components

ObjectCenter lists all the components of the current project in the Project Browser. To open the Project Browser, display the **Browsers** menu in the Main Window and select **Project Browser.**



The Project Browser shows the components of a project in two categories: files (either source or object code) and libraries. The pathnames for the libraries on your platform will be slightly different from those shown.



Instrumented object code

The makefile target loaded most of the Bounce program files in instrumented object form because this form offers the best combination of loading speed, execution speed, and run-time error checking. The Project Browser displays an **I** beside each filename to indicate it is instrumented.

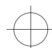
NOTE Instrumentation isn't available on all platforms. For details, read the "anomalies" section in your Platform Guide, which is usually available as an appendix to the online *ObjectCenter Reference*, or check your *Release Bulletin*.

When you instrument an object file, ObjectCenter inserts error-checking code into the file in memory and uses the error-checking code later to perform automatic run-time error checking on the file.

Trade-offs

For the fastest loading and execution speed, you can load files in regular object form, but this limits the amount of run-time error checking performed. To use the full ObjectCenter run-time error-checking capabilities, you can load a file in source form when you are actively developing it. Once a file is loaded, you can swap between object and source forms at any time.

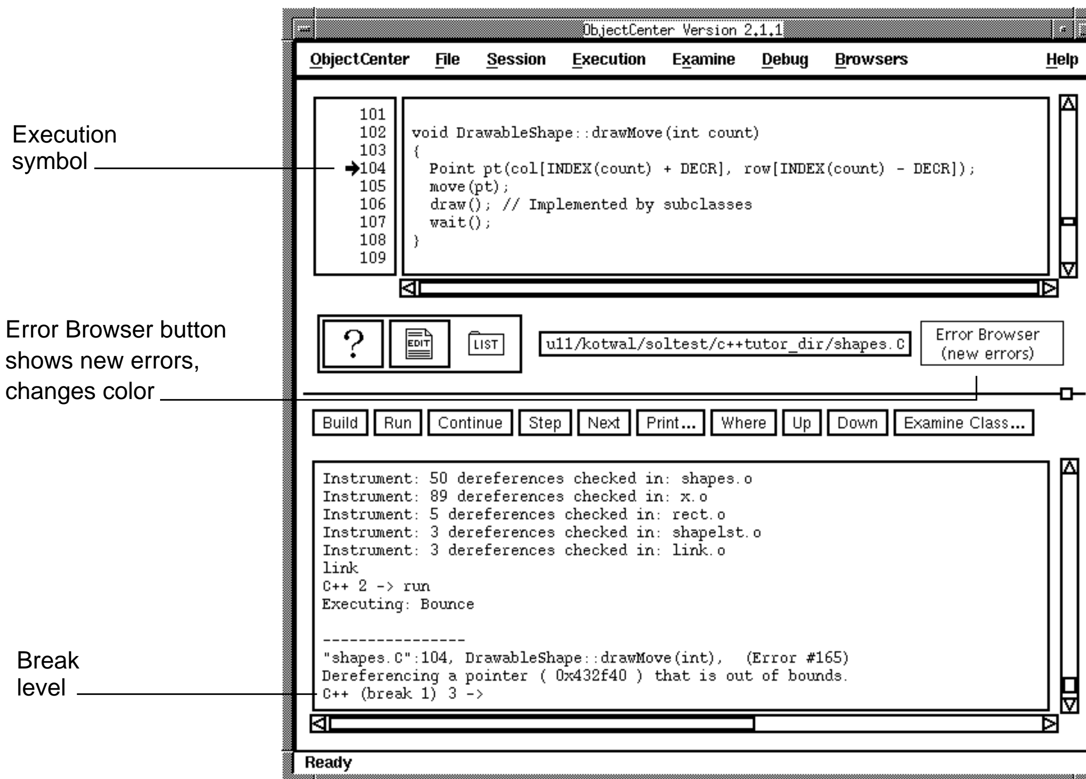
See the debugging entry in the *ObjectCenter Reference* for more details about the trade-offs when loading files in different forms.



Running the Bounce program

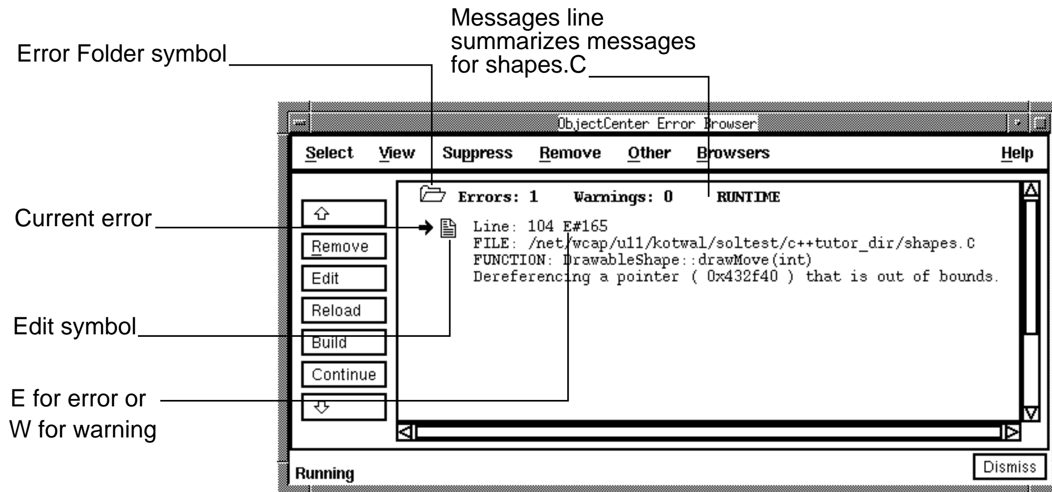
Whenever you run a program, ObjectCenter automatically checks for run-time errors in the instrumented object files and source files. To run the Bounce program, select the **Run** button in the Button panel in the Main Window.

The Run Window and Bounce window appear, and a rectangle begins bouncing in the Bounce window. ObjectCenter stops the program's execution because it detects an error.



ObjectCenter lists the source file causing the error (**shapes.C**) in the Source area and indicates the line where execution stopped (line 104) with the execution symbol. ObjectCenter also updates the **Error Browser** button in the Source area to indicate a run-time error occurred and generates a break level in the Workspace.

Open the Error Browser by clicking on the Error Browser button. The Error Browser displays a folder containing the run-time error. In this case, the Bounce program dereferenced a pointer that was out of bounds. Click on the Error Folder symbol to view the messages it contains.



A run-time error indicates a serious problem in your code, which you should correct. This error is an example of the kind of run-time error that might not be discovered in process debugging mode, but is caught by the ObjectCenter automatic run-time error checking in component debugging mode.

Swapping a file from object to source form

Using run-time error checking with instrumented object code, you have found an out-of-bounds error and narrowed down its location: line 104 of **shapes.C**. To receive a more detailed description of the error, you can swap this file from object to source form and take advantage of the more extensive run-time error checking that ObjectCenter performs on source files. Keep in mind, however, that when you load source files they run at a slower speed than object files.

To swap **shapes.o** to source form and run your program again:

1. In the Main Window, display the **File** menu and select **Swap** <**Listed File**>.

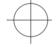
   ObjectCenter unloads **shapes.o** and loads **shapes.C**. Now you need to execute the program again to engage run-time error checking.

2. In the Main Window, select the **Run** button in the Button panel.

   ObjectCenter stops execution in the same place as before, but it displays a more informative message in the Error Browser:

   ```
   Line: 104  E#166
   FILE: .../shapes.C
   FUNCTION: DrawableShape::drawMove(int))
   Illegal array index 500 into allocated data at
   <0x513a90>. Maximum array index is 499.
   ```

   The message indicates that the array index of 500 exceeds the maximum allowed (499).



Exploring the code to understand the problem

Although you know where the run-time error occurs, you need to examine the relevant code further to understand how to fix the problem.

Displaying the value, definition, and size of variables

If you look at line 104 in the Source area, you'll see two different array indexes: **col** and **row**. To determine which index is causing the error, you need to know which one is allocated at the address specified in the error message (in this case, 0x513a90, but your address may be different).

1 Display the *value* of **col** with the **print** command:

```
C++ (break 1) 5-> print col
(short *) 0x513a90 /* (<data>) (allocated) */
```

Since the address displayed by **print** matches the one in the error message, **col** is the array index causing the error.

2 Display the *definition* of **col** with the **whatis** command:

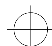
```
C++ (break 1) 6-> whatis col
short *DrawableShape::col
```

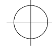
The **whatis** command shows that the **col** variable is a pointer of type **short** and a member of the **DrawableShape** class.

3 Display the *size* (as well as the name, address, and type) of **col** and *__col__ with the **info** command:

```
C++ (break 1) 7-> info col
address = 0x3084c8, name = allocated data (begins at
0x3084b0)
Size = 36, contains type: pointer.
C++ (break 1) 8-> info *col
address = 0x513a90, name = allocated data
Size = 1000, contains type: unknown.
```

The **info** command shows that the **col** array is 1000 bytes long. Since the **whatis** command indicated the elements in **col** were of type **short**, which is 2 bytes long, there are 500 elements in the array, numbered from 0 to 499.





Using shortcuts to display values and definitions

Up to this point, you've used Workspace commands to examine data, but ObjectCenter provides a faster way to examine data displayed in the Source area: pop-up windows for the **whatis** and **print** commands.

To find out why the index is set to 500, you need to examine the expression that sets the index. To do so:

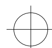
1 In the Source area, select **INDEX(count)** + **DECR** on line 104.

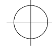
2 Press and hold the Shift key, and then press and hold the Left mouse button.

A pop-up window appears, as shown below.

```
 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                                       Print INDEX(count) + DECR
 110   void DrawableShape::bounce()        (int) 500
 111   {
```

To close the pop-up window, click the Left mouse button.





To determine how to fix the **INDEX(count) + DECR** expression, you need to know the definitions and values of **INDEX(count)** and **DECR**. To display the definition of **INDEX**:

1. In the Source area, move the mouse pointer over **INDEX** and double-click to select it.
2. Press Shift and click the Middle mouse button.

   The pop-up window shows that **INDEX** is a macro, not an ordinary variable, and it is defined as **INDEX(x) ((x) + 50)**. ObjectCenter provides you full access to macro definitions and constants as well.

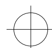
```
  101
  102  void DrawableShape::drawMove(int count)
  103  {
➔104    Point pt(col[INDEX(count) + DECR], row[INDEX(count) - DECR]);
  105    move(pt);
  106    draw(); // Implemented by subclasses
  107    wait();
  108  }
  109
  110  void DrawableShape::bounce()
  111  {

Whatis INDEX
#define INDEX(x) ((x)+50)
```

Now display the definition of **DECR**:

1. In the Source area, select **DECR**.
2. Press Shift and click the Middle mouse button.

As you can see, **DECR** is defined as a constant of 50. Thus, 50 is being added to the value of the macro **INDEX(count)**, which is 450, instead of being subtracted from it (as is the case with **row**).

```
  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()
  111  {

Whatis DECR
#define DECR (50)
```

evaloc.book : octutor1 15 Tue Jun 6 12:24:32 1995

Fixing the error and rebuilding your program

Now that you have found the error in **shapes.C**, you can use the ObjectCenter integration with your editor to correct the code.

Invoking your editor

To invoke your editor on the source code containing the error:

1. In the left margin of the Source area, move the mouse pointer over number 104 and press the Right mouse button.
2. Select **Edit line 104** from the pop-up menu with the Right mouse button.

ObjectCenter opens your editor on the source file, with the cursor placed at the beginning of the line containing the error.

Fixing the error

Now do the following to fix the error:

1. Change line 104 so that DECR is subtracted (-), not added (+), and the line reads:

   ```
   Point pt(col[INDEX(count) - DECR], row[INDEX(count)
   - DECR]);
   ```

2. In your editor, save the file.

Rebuilding your program

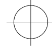
Since you have changed the source file for a component of the Bounce project, you need to rebuild the project to bring it up to date.

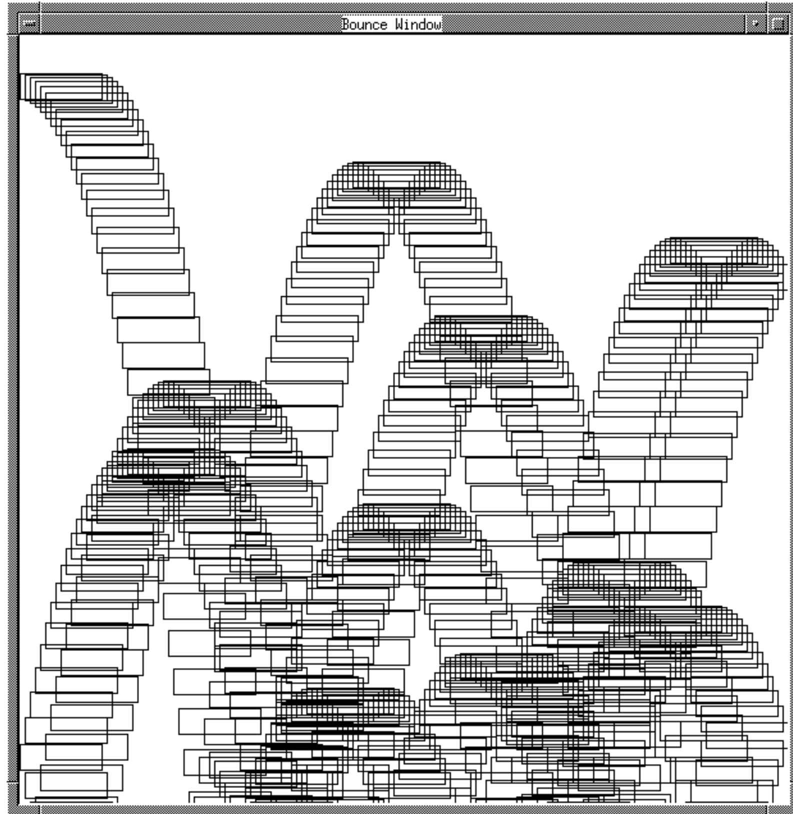
1. In the Main Window, select the **Build** button in the Button panel.

   Notice that the ObjectCenter incremental linker/loader only relinked the portion of the project that changed, which saves you significant time when you are making changes to your program. Your correction is now integrated into your project, and the modified source code appears in the Source area.

2. Select the **Run** button.

   The Bounce program runs to completion with no run-time errors, and returns you to the top level in the Workspace. The rectangle bounces in a smooth curve all the way.



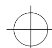


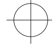
Saving a project file

You have now completed the first part of the tutorial and you can continue to Chapter 2. If you want to stop at this point, save your project before you quit ObjectCenter.

1. In the Main Window, display the **ObjectCenter** menu and select **Save Project**.
2. In the Save Project dialog box, click the **Save Project** button.

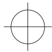
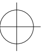
   ObjectCenter saves the project to the default filename, **ocenter.proj** in your current directory.
3. In the Main Window, display the **ObjectCenter** menu and select **Quit ObjectCenter**.
4. In the Quit Verification dialog box, click the **Quit** button.



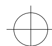


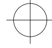
Chapter 2 Enhancing the program

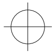
You now have a fully debugged and functioning program and are ready to enhance it. Instead of having the program bounce a rectangle, you plan to define a circle class and bounce a circle. In doing so, you learn about

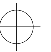



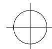
- *Using the Inheritance Browser to view the relationships among classes*
- *Adding a class to a program*
- *Using the Class Examiner to look at an individual class*
- *Looking at a program's structure with the Cross-Reference Browser*
- *Defining a class in the Workspace*
- *Setting breakpoints and stepping through code*
- *Removing breakpoints*
- *Modifying the program*









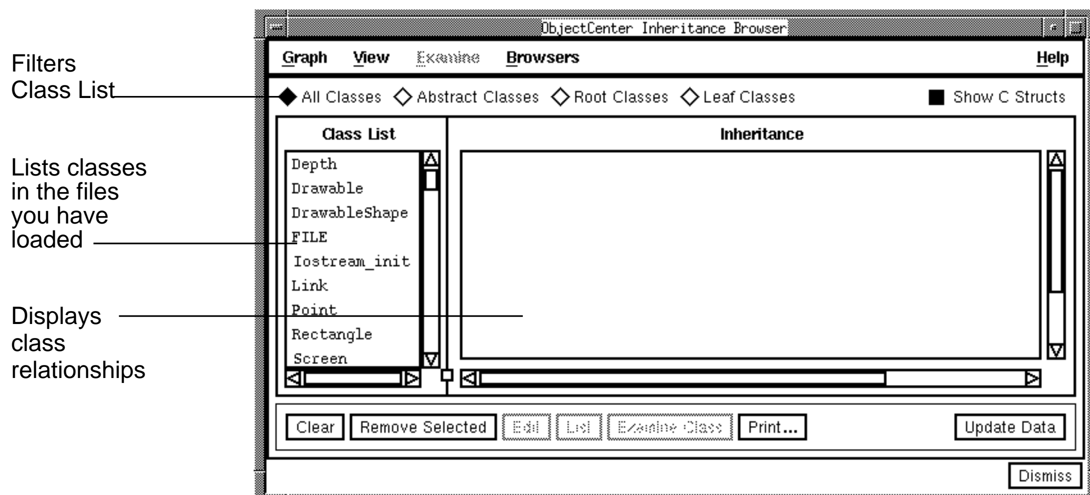


Examining classes with the Inheritance Browser

If you quit ObjectCenter at the end of the last chapter, start it again using the name of the project file you saved at the end of Chapter 1 as an argument:

```
% objectcenter ocenter.proj
```

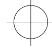
Before you add a new class to the program, you can examine the classes already in it with the Inheritance Browser. To open the Inheritance Browser, display the **Browsers** menu, and select **Inheritance Browser**.



You can use the Inheritance Browser to explore the relationships among the classes in your loaded program. The Inheritance Browser displays a list of these classes. You can filter the list to exclude names from it, and you can select names from the list and display graphical representations in the Inheritance area.

Examining the Rectangle class

Since you plan to define a circle class in order to bounce a circle with the program, examine the **Rectangle** class to see where you would add the circle class.



1 In the Inheritance Browser, select **Rectangle** from the Class List.

A class item appears in the Inheritance area for **Rectangle**. A *class item* contains the name of the class and two reference boxes: the box to the right connects to classes that are derived from this class; the box to the left connects to classes from which this class is derived.

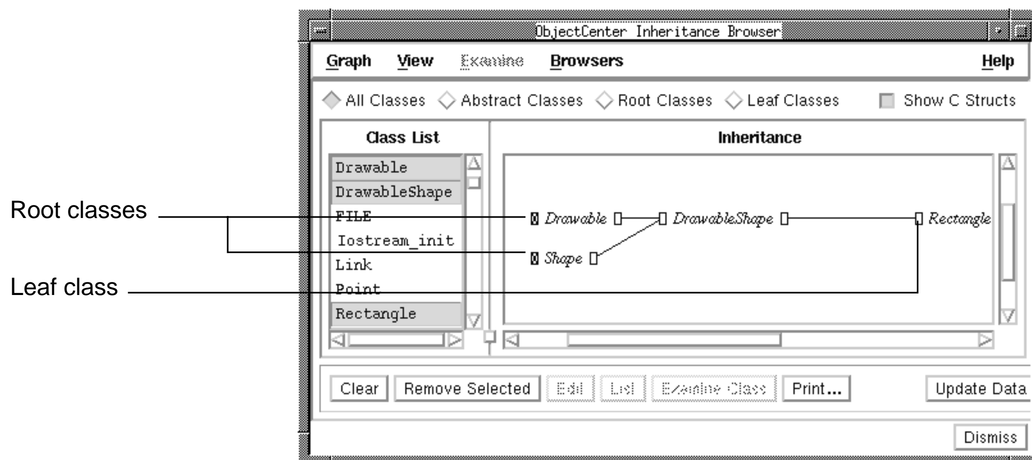
When the box to the left is filled with an X, it indicates this class is a *root class*; it is not derived from any other classes. If the box to the right is filled with an X, this is a *leaf class*; no other classes are derived from it. The **Rectangle** class item indicates that **Rectangle** is a leaf class.

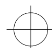
2 In the **Rectangle** class item, select the left reference box to display the class from which **Rectangle** is derived.

The **DrawableShape** class item appears in the Inheritance area. You may recall that, when you were fixing the run-time problem in the Bounce program, **col** was a data member of the **DrawableShape** class.

3 In the **DrawableShape** class item, select the left reference box to display the class from which **DrawableShape** is derived.

The **Drawable** and **Shape** class items are displayed. Both are root classes.





Adding a class to the program

Now you are ready to add the class **Circle** to the Bounce program. We have supplied an additional file, **circle.C**, with the Bounce project that defines **Circle**. To load it, type the following in the Workspace:

```
C++ 12 -> load circle.C
```

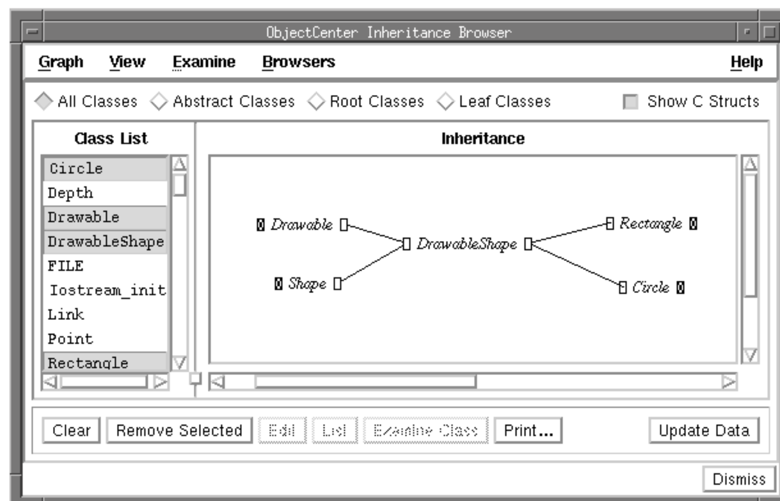
ObjectCenter loads the C++ source file **circle**.**C**, which defines and implements the class. All definitions made in the file are now available to the Bounce program.

To find out how **Circle** relates to the other classes you examined:

1 In the Inheritance Browser, select the **Update Data** button in the Button panel.

2 Select **Circle** from the Class List.

Now you see that **DrawableShape** has two classes derived from it: **Rectangle** and **Circle**. That is, rectangles and circles are both drawable shapes.

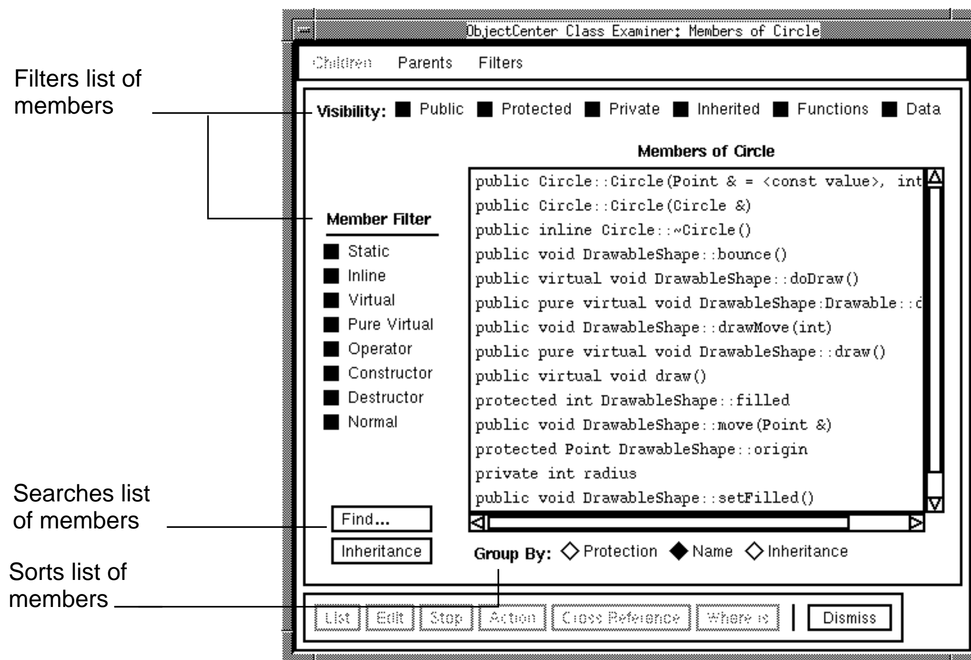


Using the Class Examiner to view individual class members

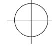
Now that you have added the Circle class to the Bounce program, you can use the Class Examiner to display the members of **Circle**. To do so:

1. In the Inheritance area in the Inheritance Browser, move the mouse pointer over the **Circle** class item.
2. Click on the item to select it.
3. Select the **Examine Class** button in the Button panel.

ObjectCenter opens the Class Examiner, as shown below:



evaloc.book : octutor2 23 Tue Jun 6 12:24:32 1995



You can use the Class Examiner to find specific kinds of members quickly. You can use the Visibility buttons to filter the list of members according to accessibility (public, protected, private), inheritance, and type of member (data or function). You can also use the Member Filter buttons to filter the list according to the type of member function (static, inline, virtual, pure virtual, operator, constructor, destructor, or normal).

Viewing protected members

Now examine **Circle**'s protected members by unselecting the **Public** and **Private** buttons.

Notice that the Class Examiner, like the rest of ObjectCenter, shows the defining class when displaying inherited members. Here, **Circle** is inheriting two data members—**origin** and **filled**—from its base class, **DrawableShape**:

```
protected int DrawableShape::filled
protected Point DrawableShape::origin
```

Viewing public uninherited members

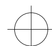
To find out which members of **Circle** are public and not inherited:

1. Unselect the **Protected** and **Inherited** buttons.
2. Select the **Public** button.

Circle contains only four public member functions and no public data members that are defined local to the class:

```
public Circle::Circle(Point&, int)
public Circle::Circle(Circle &)
public inline Circle::~Circle()
public virtual void draw()
```

Notice that **draw()** is a virtual function. ObjectCenter shows you that each class implements its own version of **draw().** That is, **draw()** draws a **Rectangle** differently from a **Circle**, even though each is a **DrawableShape.**



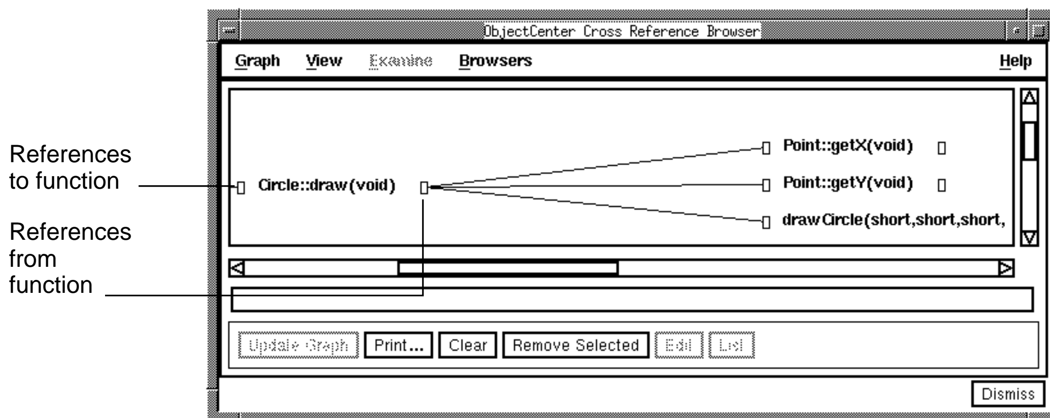
Looking at the program's structure in the Cross-Reference Browser

Since drawing the shape is a central part of the Bounce program, you want to understand how the **draw()** virtual function is called from within the program and used by different classes. Using the Cross-Reference Browser, you can easily see where functions and variables are used in a program. To examine the calling structure for **draw()**:

1 In the Class Examiner, select **draw()** from the list of members.

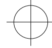
2 Select the **Cross Reference** button in the Button panel.

ObjectCenter displays the Cross-Reference Browser:

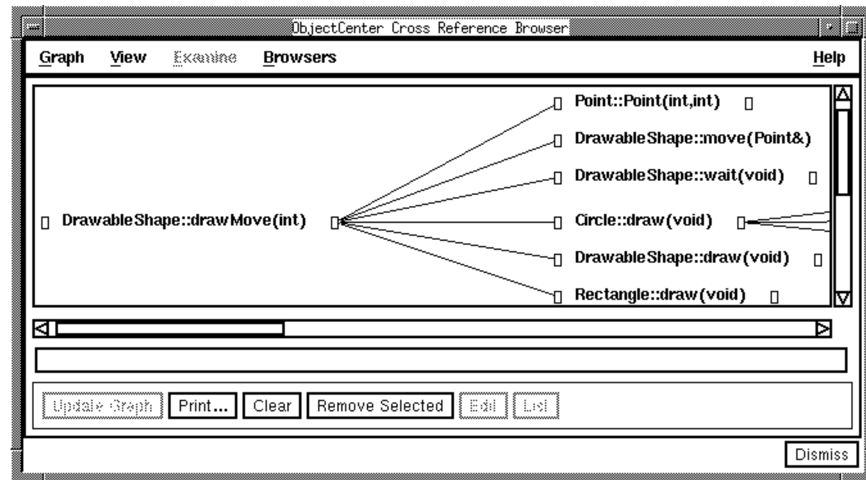


The Cross-Reference Browser displays a *cross-reference item* for **draw()**. This item displays the name of the function, the class of which it is a member, the data type of the arguments passed to the function, and two reference boxes. The box to the right connects to functions that are called by **Circle::draw()**; the box to the left connects to functions that call **Circle::draw()**.

When the box to the left is filled with an X, it indicates this function is not called by any other function (such as **main()**). If the box to the right is filled with an X, this function calls no other functions. In this case, **Circle::draw()** calls three functions: **getX()**, **getY()**, and **drawCircle()**.



If you scroll to the left, you see that **DrawableShape::drawMove()** calls **Circle::draw()**. To find out the other drawing functions that are called by **DrawableShape::drawMove()**, select the right reference box in the **DrawableShape::drawMove()** cross-reference item.



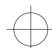
The Cross-Reference Browser shows you that **drawMove()** calls five other functions, including **Rectangle::draw()** and **DrawableShape::draw()**. Therefore, each class does implement its own version of **draw()**.

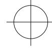
Following the drawMove() function back to main()

Now follow the references to **DrawableShape::drawMove()** all the way back to the **main()** routine:

1. In the **DrawableShape::drawMove()** cross-reference item, select the left reference box.
2. In the **DrawableShape::doDraw()** cross-reference item, select the left reference box.
3. In the **DrawableShape::bounce()** cross-reference item, select the left reference box.

You can now see that **main()** calls **bounce()**, which in turn calls **doDraw()**. The **doDraw()** function is a virtual function just like **draw()**; that is, each class derived from **DrawableShape** can implement its own version of **doDraw()**. If you select the right reference box in the **DrawableShape::doDraw()** cross-reference item, you see that, in the Bounce program, there is only one version of **doDraw()**, that of **DrawableShape**.





Prototyping your enhancements

Now that you've examined the class and member function relationships within the Bounce program, you can prototype your enhancement before modifying your code. That is, you can test the implementation of the new **Circle** class, by creating an instance of a circle and bouncing it.

Close or iconify any Browsers you have opened, and return to the Workspace in the Main Window.

Defining a class instance in the Workspace

So far, you have used the Workspace to enter ObjectCenter commands, such as **whatis** and **print**. When you are in component debugging mode, as you are now, the ObjectCenter Workspace is also a full C++ interpreter. You can type C++ statements in the Workspace, and ObjectCenter executes the statements dynamically.

We will now use the Workspace to create a circle, then bounce it. The **Circle** class has a constructor that provides default initialization values, so you can create a circle without providing initialization values.

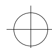
Creating a circle

To create a circle named **c1**, do the following. Be sure to include the semicolon at the end of the statement to indicate that you have completed the statement. If you do not, ObjectCenter shows that it is waiting for you to complete the statement by giving you a continuation prompt (such as `C++ 17 +>`).

```
C++ 15 -> Circle *c1;
C++ 16 -> c1 = new Circle();
(class Circle *) 0x4c3018 /* (class Circle)
(allocated) */
```

ObjectCenter creates **c1**, as echoed in the Workspace.

Although you are just declaring a variable here, a lot of code might be executed when you create a C++ class instance. For example, constructors need to be called. In fact, because you can overload operators and functions, there are many situations in C++ when it might be difficult to see exactly what code needs to be executed in order to carry out a particular statement.



Setting a breakpoint

To follow the execution of the code, set a breakpoint and step through the code as the program bounces the circle you just created. When ObjectCenter reaches a breakpoint during execution, it stops execution and indicates the current line of code in the Source area. You can then explore the state of the program and step through your code a line at a time to see exactly what is going on.

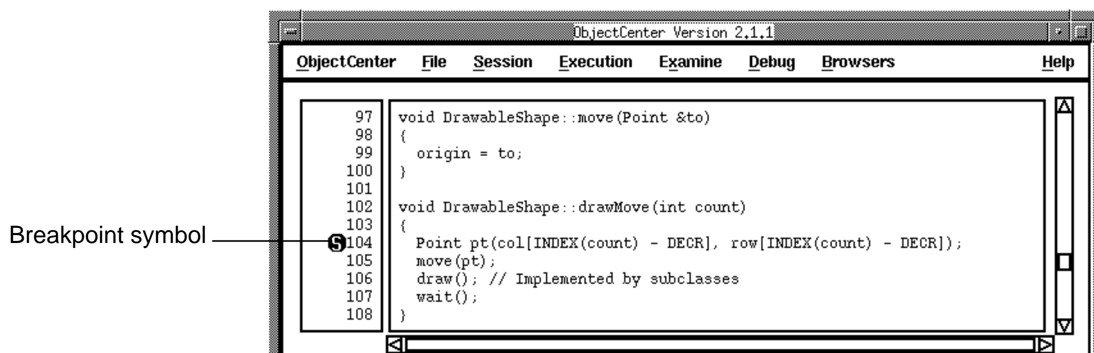
In the Cross-Reference Browser, you examined the **drawMove()** function, which is one of the functions called when an object is bounced. To set a breakpoint on that function so that whenever the function is called, ObjectCenter stops execution:

1 In the margin of the Source area, move the mouse pointer over the number 104.

2 Press the Right mouse button to display the pop-up menu.

3 Select the **Set Breakpoint Here** menu item.

A breakpoint symbol appears to the left of the line number.



As a shortcut, you can set a breakpoint by selecting the line number instead of using the pop-up menu.

Bouncing the newly created circle

To bounce the circle, call **bounce()**, which is a member function in **Circle** and inherited from **DrawableShape**. You can call the member function interactively in the Workspace in the same way you would call the function in a program:

```
C++ 19 -> c1->bounce();
```

ObjectCenter displays the Bounce window with nothing in it, then stops in **drawMove()**. To indicate that the program is stopped at a breakpoint, ObjectCenter places the execution arrow next to the breakpoint symbol.

```
ObjectCenter Version 2.1.1
ObjectCenter  File  Session  Execution  Examine  Debug  Browsers  Help

  97   void DrawableShape::move(Point &to)
  98   {
  99     origin = to;
 100   }
 101
 102   void DrawableShape::drawMove(int count)
 103   {
→S104    Point pt(col[INDEX(count) - DECR], row[INDEX(count) - DECR]);
 105     move(pt);
 106     draw(); // Implemented by subclasses
 107     wait();
 108   }
```

Stepping through the code

To step through the code by executing one statement at a time:

1 In the Button panel, select the **Step** button. The Bounce program defines a **Point** by calling the **Point** constructor.

```
ObjectCenter Version 2.1.1
ObjectCenter  File  Session  Execution  Examine  Debug  Browsers  Help

  25       *       Implementation of Point class                      *
  26       *                                                          *
  27       *----------------------------------------------------------*/
  28
  29   Point::Point(int X, int Y)
 →30   {
  31     x = X;
  32     y = Y;
  33   }
  34
  35   Point::Point(Point &anotherPoint)
  36   {
```

2 Select the **Step** button four times (three on some platforms) to step through the **Point** constructor until ObjectCenter returns to **drawMove()**, at a call to **move()**.

evaloc.book : octutor2 29 Tue Jun 6 12:24:32 1995

3 Select the **Next** button to step over **move().**

Execution is now stopped at **draw()**. As mentioned earlier, **draw()** is a virtual function; each class in the **DrawableShape** hierarchy implements its own version of **draw()**. For example, a **Circle** is drawn differently from a **Rectangle**.

4 Select the **Step** button to step into **draw()**.

When **draw()** is called, ObjectCenter dynamically determines which **draw()** is appropriate.

Since you are bouncing a **Circle**, ObjectCenter calls **Circle**::**draw()**, as shown below. If you were bouncing a **Rectangle**, ObjectCenter would have called **Rectangle**::**draw()**.

ObjectCenter Version 2.1.1

ObjectCenter File Session Execution Examine Debug Browsers Help

```
 7        DrawableShape(origin) ,radius(itsRadius)
 8  {
 9  }
10
11  Circle::Circle(Circle& c) : DrawableShape(c), radius(c.radius)
12  {
13  }
14
15  void Circle::draw()
16  {
➔17   drawCircle(origin.getX(), origin.getY(), radius, filled);
18  }
```

5 Select the **Continue** button.

ObjectCenter continues execution until the Bounce program stops at the breakpoint on line 104 again. You now have a circle in the Bounce window.

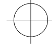
Removing a breakpoint

Now that you are sure that your enhancement draws a single circle, remove the breakpoint and continue the program to see if it bounces the circle to completion:

1 In the Source area, select the breakpoint symbol to remove it.

2 In the Button panel, select the **Continue** button.

The circle completes its bouncing.



Modifying the program

Now that you've prototyped the **Circle** class and it bounces correctly, you can modify the source code to bounce a circle instead of a rectangle. Make the following changes to the beginning of the **main()** function in **main1.C**.

Changing the source code

1 Invoke your editor on **main1.C**:

```
C++ 20-> edit main1.C
```

2 Define an instance of **Circle**. Place it below **r = new Rectangle(P1, P2)**:

```
Circle *c;
c = new Circle();
```

The first part of the **main()** function in **main1.C** should now look like this:

```
Point P1(50, 50);
Point P2(64, 20);
Rectangle *r;
r = new Rectangle(P1, P2);

Circle *c;
c = new Circle();
```

3 On the next line, change the call to **r->bounce()** so it bounces the circle (**c**) you just defined:

```
c->bounce();
```

The variable **r** is the rectangle that the program bounced. Now you are specifying that the newly created circle, **c**, should be bounced.

Saving, building, and running

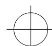
Now save the changes, reload **main1.C**, and run the modified program:

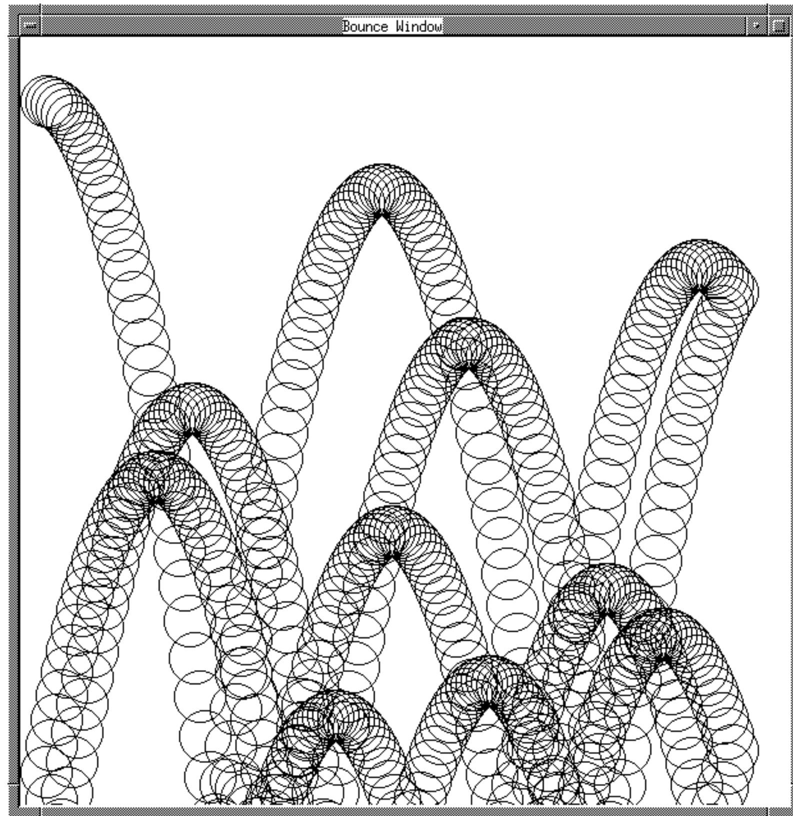
1 In your editor window, save the file.

2 In the Main Window, select the **Build** button in the Button panel.

3 In the Button panel, select the **Run** button.

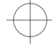
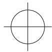
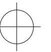
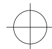
You have successfully modified the Bounce program — it now bounces a circle instead of a rectangle.

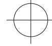




Saving your project file

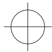
You have completed the second part of the tutorial and you can continue to Chapter 3. If you want to stop at this point, save your project before you quit ObjectCenter as described in 'Saving a project file' on page 16.

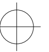







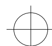
Chapter 3 Adding template code to the program

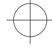
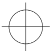
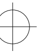
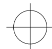
Now that you've modified the Bounce program to bounce a circle, you can improve this enhancement to bounce circles of different sizes. By using class templates and the ObjectCenter automatic template instantiation system, you can create a linked list of circles of different sizes. In doing so, you'll learn about

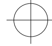




- *Instantiating a class template in the Workspace*
- *Using the expand command*
- *Loading code that uses templates*
- *Examining your code with the Data Browser*





Starting with Chapter 3

If you have just completed Chapter 2, continue on page 36. If you haven't worked through the previous chapters, follow these steps before you begin Chapter 3:

1. Install the examples as described on page viii if you haven't already done so and change directory to the **c++tutor_dir** directory.
2. Invoke ObjectCenter with the **objectcenter** command.
3. Make the **bounce_project** target:

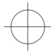
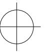
```
c++ 1-> make bounce_project
```

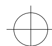
4. Modify the **shapes.C** file as described in 'Fixing the error' on page 15 and select the **Build** button in the Button Panel.
5. Load **circle.C:**

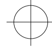
```
c++ 2-> load circle.C
```

6. Create an instance, **c1**, of the Circle class in the Workspace:

```
C++ 3-> Circle *c1;
C++ 4-> c1 = new Circle();
```





Instantiating a class template in the Workspace

The **c++tutor_dir** directory already contains the **List.h** file, which provides the definitions necessary to create a linked list of circles from a template. Specifically, you can create the list as an instantiation of the **List** class template with **Circle** as the parameterized type.

Prototyping your enhancement in the Workspace

As with your previous enhancement, you prototype it in the Workspace before modifying the source code.

1 Load in the definitions from **List.h**. By default, ObjectCenter loads header files with the **-dd=off** load switch. This switch generates all code, even if it is not used.

```
C++ 31 -> load_header List.h
Loading (C++): List.h
```

You can use either **load** or **load_header** to load local header files like **List.h**. You should always use the **load_header** command to load a system header file without specifying its path, or to load interdependent header files into a single module. For more information, see the **load_header** entry in the *ObjectCenter Reference.*

2 Create an object named **CircleList** from the **List** class template and initialize the list using **c1**, the circle you created previously in the Workspace.

```
C++ 32 -> List<Circle> CircleList(*c1);
List__pt__8_6Circle____ct.c:
Loading: ptrepository/List__pt__8_6Circle____ct.o
List__pt__8_6Circle____data.c:
Loading: ptrepository/List__pt__8_6Circle____data.o
(class List<Circle> *) 0x3dd8c0 /* (class
List<Circle>) CircleList */
```

Examining the List template class

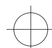
Now that you have instantiated the **List** template class as a **CircleList**, examine the class with the Class Examiner:

1 Open the Inheritance Browser.

2 Select the **Update Data** button in the Button panel.

3 In the Class List, select **List<Circle>**.

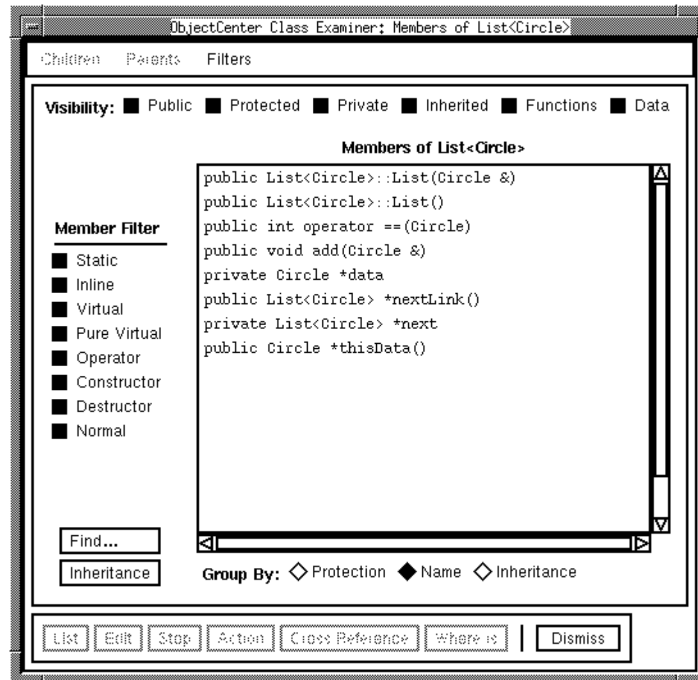
4 In the Inheritance area, select the **List<Circle>** class item.



5 In the Button panel, select the **Examine Class** button.

6 In the Class Examiner, make sure that *all* the visibility buttons are selected (for example **Public, Protected**, **Private**).

The Class Examiner displays all the members of **List**<**Circle**>.



Notice that **List** classes have an **add** member function for adding circles to the list. Before you can add circles to the list, however, you need to create the circles.

Creating circles of different sizes

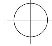
To find out how you create circles of different sizes, open the Class Examiner for the **Circle** class and look at the constructor functions available:

1 Open the Inheritance Browser.

2 In the Class List, select **Circle**.

3 In the Inheritance area, select the **Circle** class item.

4 In the Button panel, select the **Examine Class** button.



5 In the Member Filter area, unselect all buttons except for **Constructor**.

The Class Examiner shows two different constructors:

```
public Circle::Circle(Point &, int)
public Circle::Circle(Circle &)
```

The first constructor creates new circles and takes a **Point** and an **int** as arguments. The second constructor, a copy constructor, takes a **Circle** as an argument. Earlier, you used the default constructor to create the circle **c1**. ObjectCenter generates the default constructor when you don't explicitly provide a constructor.

Creating a Point in the Workspace

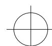
Find out more about the **Point** class with the **whatis** command, and then create a **Point** in the Workspace:

```
C++ 34 -> whatis Point
class Point {
 int x;
 int y;
} ;
C++ 35 -> Point pt(120,20);
(class Point *) 0x306f00 /* (class Point) pt */
```

Creating two more circles in the Workspace

Now that you have created a **Point** and know how to create circles and add them to the list, create two more circles named **c2** and **c3** and use **List** member functions to add **c2** and **c3** to the list:

```
C++ 36 -> Circle *c2;
C++ 37 -> c2 = new Circle(pt, 90);
(class Circle *) 0x30a688 /* (class Circle)
(allocated) */
C++ 38 -> CircleList.add(*c2);
List__pt__8_6Circle__add.c:
Loading: ptrepository/List__pt__8_6Circle__add.o
(void)
C++ 39 -> Circle *c3;
C++ 40 -> c3 = new Circle(pt, 120);
(class Circle*) 0x30a708 /* (class Circle)
(allocated) */
C++ 41 -> CircleList.add(*c3);
(void)
```

Using the expand command

In this example, **Circle::Circle**, the constructor for a circle, is an overloaded function. To disambiguate overloaded functions and operators, use the **expand** command to show which functions could be called if a section of code is executed. The set of functions *actually* executed during a given run depends on run-time conditions and might be a subset of the functions listed by **expand**.

Using the expand command to evaluate statements

To see how a circle would be created when the constructor is called with a **circle** as an argument instead of a **Point** and an **int**:

```
C++ 42 -> expand Circle c(*c1)
Circle::Circle(Circle &)
inline Circle::~Circle()
C++ 43 -> expand Circle c(pt,9)
Circle::Circle(Point &, int)
inline Circle::~Circle()
```

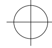
The **expand** command *evaluates* the selected statements; it does not execute them. No functions are called and no program values are changed by the evaluation of the statements. Thus, in the previous example, the circle named **c** is not created.

Testing your prototype

To test your prototype, try to bounce the second item on the list:

```
C++ 44 -> CircleList.next->thisData()->bounce();
List__pt__8_6Circle__thisData.c:
Loading: ptrepository/List__pt__8_6Circle__thisData.o
```

ObjectCenter creates a Bounce window and bounces a large circle.



Loading code that uses templates

Since your prototype works, you are ready to modify the Bounce program. To do so, you need to add the code you just tried in the Workspace to the **main()** function. You also need to include the code in **List.h**.

When you edit the **main()** function, however, you make one change from the prototype. Instead of bouncing just one circle as you did in the Workspace, you use a loop to walk through the list of circles and bounce each one.

To avoid your having to retype all this code, you can load the **main2.C** file from the **c++tutor_dir** directory. Since you're actively developing this code, load it as source rather than as instrumented object code:

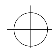
```
C++ 45 -> unload main1.o
Unloading: main1.o
C++ 46 -> load main2.C
Loading (C++): main2.C
```

NOTE Notice that you *do not* have to load or compile the **List.c** file, which contains the definitions for the **List** class template declarations in **List.h**. The automatic instantiation system takes care of finding and compiling the necessary definitions for **List**.

See the templates entry in the *ObjectCenter Reference* for more information about the automatic template instantiation process.

Now run the new Bounce program: In the Main Window, select the **Run** button in the Button panel.

ObjectCenter creates three Bounce windows and bounces increasingly larger circles in each.



Examining your code with the Data Browser

Now that you have created a linked list using the **List** class template, you can use the Data Browser to examine the various elements on the list dynamically.

Looking at the linked list of circles

Set breakpoints in the Bounce program where each element is added to the list. Then, as you step through the code, you can use the Data Browser to display the list and watch elements as they are being added.

1 Return to the Main Window to list your program:

```
C++ 48 -> list main
Listing file 'main2.C', line 1 ...
```

2 Set breakpoints on line 19, 20, and 21 in **main2.C**.

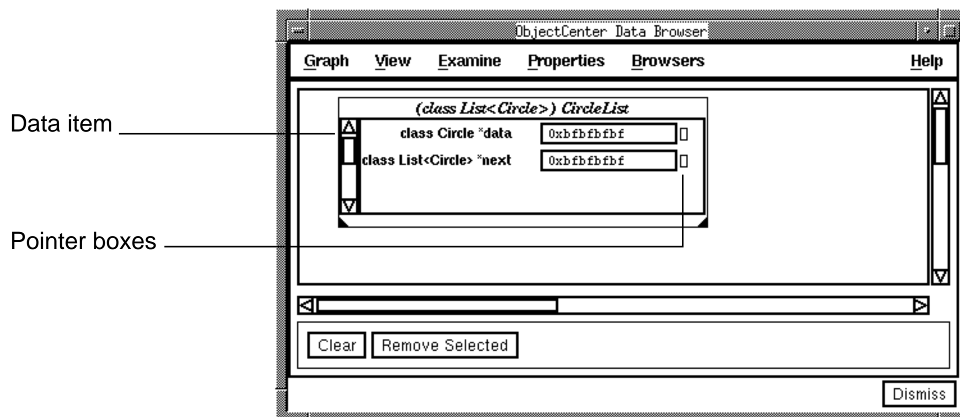
3 Select the **Run** button.

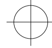
ObjectCenter begins execution of the program and stops at line 19, the first breakpoint.

4 Select **CircleList** on line 19.

5 Display the **Examine** pulldown menu and select **Display<Selection>**.

ObjectCenter opens the Data Browser and displays an empty data item for **CircleList**.



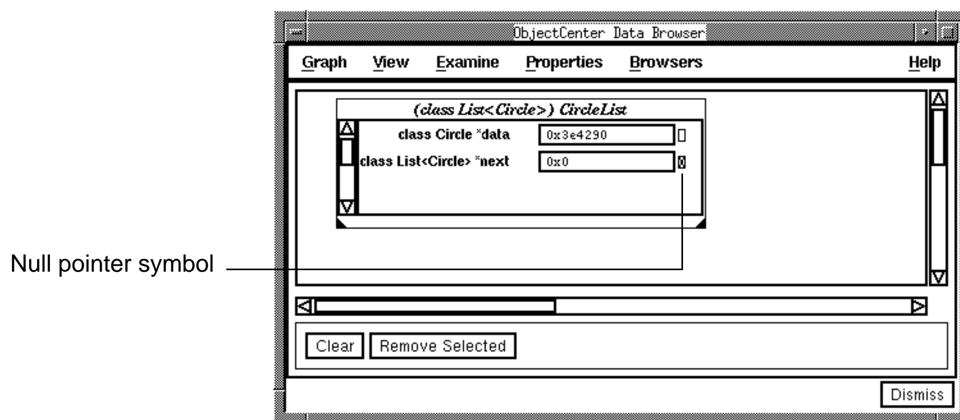


A **data item** displays the class, name, and value of the variable. If the variable contains other data, a list of the names and values are also displayed in the data item. Pointer boxes indicate pointers to other data. The **CircleList** data item contains two pointers: **data** and **next**. Since the pointer boxes for **data** and **next** are empty and the addresses displayed do not appear to be valid, these pointers are uninitialized.

To see what happens to these pointers next:

1. In the Main Window, select the **Continue** button in the Button panel.
2. Display the **Browsers** menu and select **Data Browser** to raise the Data Browser.

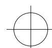
Watch how ObjectCenter displays the changes in the program's data structure dynamically as the program executes. At the current breakpoint, which is on line 20, the Bounce program has initialized the **CircleList** object with the first circle's data. Notice that the addresses next to the **data** and **next** members of the object have changed. The pointer box for **next** has an X in it, which indicates it is a null pointer, because there is only one circle now on the list.

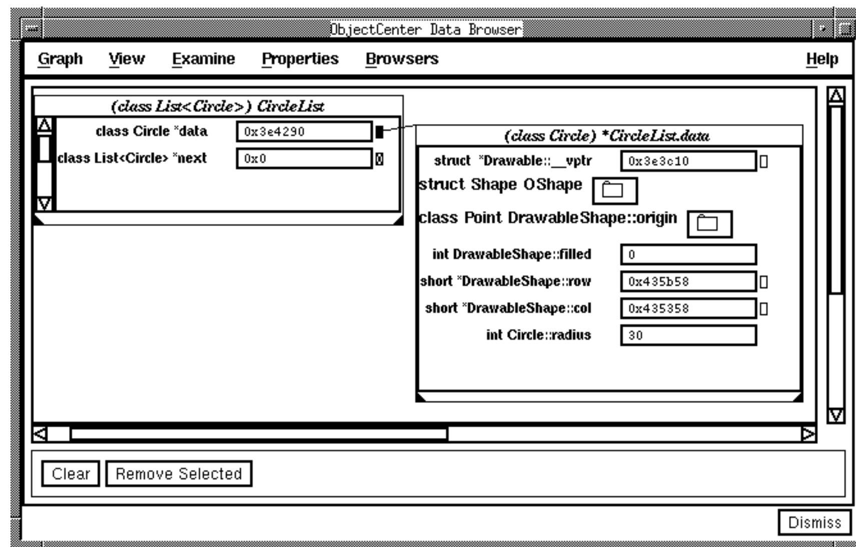


Dereferencing a pointer

When a pointer has been allocated, you can dereference it by selecting its pointer box. To display the contents of the **data** member of **CircleList**, dereference its pointer, and then select the pointer box beside **data**.

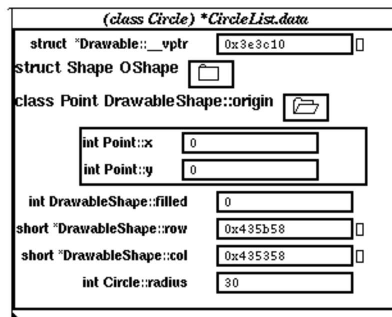
The Data Browser displays the data item for **CircleList.data** and connects it to the pointer box from **data**. As you can see, the first circle has a radius of 30.





Also notice the folder symbol next to **origin**, which indicates a nested array or structure. To display the contents of **origin**, select the folder symbol beside **origin**.

The Data Browser opens the **origin** folder and displays two integers: **x** and **y**.



Examining data while continuing execution

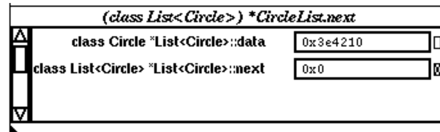
To see what happens at the next breakpoint on line 21:

1. In the Main Window, select the **Continue** button.

   Now the program has added a second circle to the **CircleList** object, as shown by the address for the **next** data member and its empty pointer box.

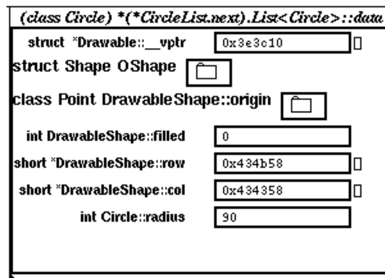
2 Select the pointer box beside the **next** member of **CircleList**.

The Data Browser displays **CircleList.next**.



3 Select the pointer box beside the **data** member of **CircleList.next**.

The Data Browser displays the second circle on the list. Notice that it has a radius of 90, as specified on line 17.



Completing execution

To complete the execution of the program and leave the Data Browser:

1 In the Data Browser, select the **Dismiss** button.

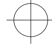
2 In the Main Window, select the **Continue** button.

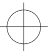
ObjectCenter closes the Data Browser and completes execution of the Bounce program by bouncing three circles.

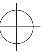
Continuing the tutorial

You have now completed the main part of the tutorial. Thus far, we have relied on a makefile to load the Bounce program. If you would like to practice loading code into ObjectCenter without using a makefile, go on to 'Using the load command' on page 45. If you would like to try out the precompiled header facility, complete 'Speeding up compilation with precompiled headers' on page 53.

To try loading your own code into ObjectCenter, go to 'Loading your application' on page 61.



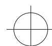


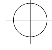
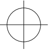
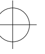
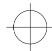


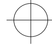
Chapter 4 Using the load command

In this chapter, we load a version of the Bounce program into ObjectCenter using the load command.

This version of the program bounces a linked list of circles.





Starting Chapter 4

To begin this chapter:

1. If ObjectCenter is already running, exit your session as described in 'Saving a project file' on page 16.
2. If you haven't already created the tutorial directory, install the examples as described in 'Setting up the tutorial directory' on page viii.
3. Go to the tutorial directory and invoke ObjectCenter:

```
% cd ~/c++tutor_dir
% objectcenter
```

Output when loading object files

When you load object files in this chapter, the output of the **load** command will depend on whether you have already completed the previous chapters.

If your **c++tutor_dir** directory already contains object files for the source files used in this chapter, ObjectCenter loads the object files. For example:

```
C++ 1 -> load shapes.o
Loading: shapes.o
```

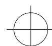
If the object files don't exist in **c++tutor_dir**, ObjectCenter creates them. For example:

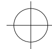
```
C++ 1 -> load shapes.o
Cannot open '/net/test/c++tutor_dir/shapes.o'.
No such file or directory
Executing:/net/.../CC /net/test/c++tutor_dir/shapes.C
/net/test/c++tutor_dir/shapes.C:
Loading: shapes.o
```

Load errors

If ObjectCenter finds errors when you load a file, it immediately unloads it and reports errors in the Error Browser. For example, if you try to load **List.c**, which is a template definition file, you see output like this:

```
C++ 2 -> load List.c
Loading (C++): List.c
Unloading: List.c
*** Check error browser for more details. ***
Warning: 1 module currently not loaded.
C++ 3 ->
```



Setting options

Before loading files, you need to set the **program_name** and **load_flags** options. The **load_flags** option specifies the switches that should be appended to the **load** command when you invoke it without any switches.

Loading files into ObjectCenter is analogous to compiling files with your compiler. When loading files, you want to pass the same switches (such as -**I**, -**D**, and -**L**) that you normally pass to the compiler. If you don't specify the correct switches, ObjectCenter may not find the correct header files or libraries when loading.

1. Set the **program_name** option:

```
C++ 1-> setopt program_name bounce
```

2. Set the **load_flags** option to include all the switches that are passed to the compiler, as specified by **$(FLAGS)** in the makefile. The -**I** and -**L** switches specify the list of directories that the **load** command should search for header files and libraries (respectively). The **load_flags** option saves you from specifying these switches each time you load a file in your project. Here are some typical values for the **load_flags** option:

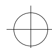
   For Sun™ platforms:

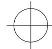
```
C++ 2-> setopt load_flags -dd=off
-I/usr/include/X11R4 -I/usr/include/X11R5
-I#$OPENWINHOME/include -L/usr/lib/X11R4
-L/usr/lib/X11R5 -L#$OPENWINHOME/lib
```

   For HP™ platforms:

```
C++ 2-> setopt load_flags -dd=off
-I/usr/include/X11R4 -I/usr/include/X11R5 -I/include
-L/usr/lib/X11R4 -L/usr/lib/X11R5 -L/lib
```

There are many other options that affect ObjectCenter commands and windowing options. They're discussed briefly on page 95, and in more detail in the options entry in the *ObjectCenter Reference.*





Loading the Bounce program

We load the files for the Bounce program in several different forms.

Loading object files with debugging information

Begin by loading the files that you plan to debug as regular object files with debugging symbols.

If the object file does not exist or is out of date

When you load object files, ObjectCenter does not use most of the switches you specify until the file needs to be recompiled or you swap it to source. If ObjectCenter cannot find an object file specified with the **load** command or the file is out of date relative to its source file, ObjectCenter attempts to rebuild the file.

Load the **main2.o** object file with debugging symbols:

```
C++ 3 -> load main2.o
Cannot open ‘main2.o’
No such file or directory
Executing make main2.o
CC ... -I/s/apps/openwin3.0/include +d -g -c main2.C
main2.C:
Loading main2.o
```

In this example, a makefile exists in the same directory as the object file, so ObjectCenter invokes the **make** utility to compile the object file. If the makefile did not exist or could not be found in any of the directories specified by the **path** option, ObjectCenter would have invoked the compiler directly.

Load **shapes.o** as a regular object file with debugging symbols:

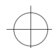
```
C++ 4 -> load shapes.o
```

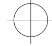
Loading object files without debugging information

Typically, you load most of the files as regular object files without debugging information (with the -**G** switch). This improves the load-time and run-time performance of ObjectCenter. You also attach the X11 library with the -**l** switch.

Load the **link.o**, **rect.o**, and **x.o** files and attach the **X11** library:

```
C++ 5-> load -G link.o rect.o x.o -lX11
```



In the example, you load the X11 library with the -**G** option even though the X11 library doesn't have debugging symbols. We recommend that you load your own libraries with -**G** unless you are specifically debugging the library.

Loading C files

By default, ObjectCenter assumes both object files and source files are C++ modules. To load a C object file or source file, you need to use the -**C** switch.

Load the **table.o** object file as a C file:

```
C++ 6 -> load -C table.o
```

Loading source files

Since Bounce is a small program, you can also load the files that you are debugging as source files. With large applications, we recommend that you swap an object file with a source file when you are tracking down a specific error.

Load **shapelst.C** as a source file:

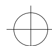
```
C++ 7 -> load shapelst.C
```

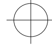
Loading header files

By default, ObjectCenter uses demand-driven code generation and loads code only if it is used for most files. However, header files are loaded with demand-driven generation turned off by default. In this instance, you can use either the **load** command or the **load_header** command to load **List.h**, because it's in your current directory. The **load_header** command is particularly useful if you want to load a system header file, or if you want to load several interdependent header files into a single module.

Load the **List.h** header file:

```
C++ 8 -> load_header List.h
```



Linking from libraries

When you have loaded all the files you think you need, issue the **link** command to link modules from the libraries.

```
C++ 9 -> link
```

When you link your project, ObjectCenter incrementally links the individual library modules as they are needed by your application, instantiates any templates that are defined, and lists any undefined symbols. In this case, ObjectCenter issues this message:

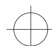
```
Loading: ptrepository/List__pt__8_6Circle____ct.o
...
Undefined symbols:
extern void draw_circle();
Circle::Circle()
    extern Unknown DefaultPt;
```

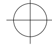
You neglected to load the **circle.o** file, so you need to load it and link again:

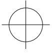
```
C++ 10 -> load circle.o
Loading (C++): circle.o
C++ 11 -> link
```

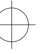
The Bounce program is now loaded into the ObjectCenter environment. You can run the program by selecting the **Run** button in the Button Panel, and you can use the techniques described in the rest of this guide to examine and enhance it.

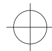
For complete information on the **load** and **link** commands, refer to the **load** and **link** entries in the *ObjectCenter Reference*.

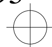






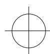


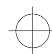




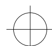
Chapter 5 Speeding up compilation with precompiled headers

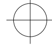
When you have enhanced your program successfully, you can use the precompiled header file facility in the ObjectCenter C++ translator to speed up compilation. This facility keeps track of header files that have been compiled, avoids recompiling them unnecessarily on subsequent compilations of the same program, and thus speeds up the compilation considerably.

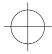


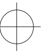


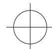
For this chapter we provide a very simple example in the c++tutor_dir directory that you can use to see how precompiled header file skipping works.











Creating the skip information file

X Window System applications typically use many large header files. Therefore, since the **x.C** module implements the X Window System support in the Bounce project, it is a good candidate for compiling with header-file skipping.

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

1 If you haven't already created the tutorial directory, install the examples as described on page viii.

2 Invoke ObjectCenter:

```
% objectcenter
```

3 List **x.C** in the Source area:

```
C++ 1 -> list x.C
```

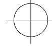
Notice that **x.C** includes seven global header files and two local header files. In general, we recommend using global header files for header skipping rather than local header files.

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

```
C++ 2 -> list skip
```

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

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



Recompiling with header-file skipping

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

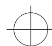
To recompile **x.C**, do the following:

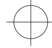
```
C++ 3 -> make skipping
```

As part of the ouput, you should see a series of four timestamps and three compiles. The first compile does not use header-file skipping, the second compile creates the repository for the precompiled header files, and the last compile uses header-file skipping, restoring the files from the repository instead of recompiling them.

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

The time taken for each compile will depend on the configuration of your network and system. In this sample run, the normal compilation took 41 seconds and the compilation with header-file skipping took 25 seconds. The initial creation of the header-file repository took 95 seconds. With more complex programs that use large numbers of header files, the speed improvement can be dramatic.





A more complex skip information file

Here's a more complex skip information file (this example isn't available online). This file is designed to be used for a large project where many files **#include** one or more of the following header files: **iostream.h**, **stdlib.h**, **math.h**, and **OI/oi.h**.

```
<iostream.h> <stdlib.h> <math.h> <OI/oi.H> -hdrepos /export/utils/HR
<iostream.h> <stdlib.h> <OI/oi.H> -hdrepos /export/utils/HR
<iostream.h> <OI/oi.H> -hdrepos /export/utils/HR
<iostream.h> <stdlib.h> <math.h> -hdrepos /export/utils/HR
<iostream.h> <stdlib.h> -hdrepos /export/utils/HR
<iostream.h> -hdrepos /export/utils/HR
```

Suppose you have a file, **test1.C**, which begins with these **#include** directives:

```
#include <iostream.h>
#include <stdlib.h>
#include "test1.h"
```

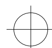
To use the save-and-restore mechanism, compile **test1.C** with the +**k** switch specifying the skip information file above, which we assume resides in **/export/utils/HR.cf**. :

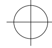
```
% CC -g +k=/export/utils2/HR.cf -I/path1 -I/path2 test1.C
```

The first time **test1.C** is compiled, an image is created in a repository in **/export/utils/HR**, the publicly accessible area specified with -**hdrepos** in **/export/utils/HR.cf**. The longest set of header files found in the skip information file is used, in this case:

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

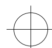
After the first compile, an image is restored from this repository whenever anyone on the project uses the +**k=/export/utils2/HR.cf** switch to compile any file beginning with the same ordered pair of header files. Additional images are saved in the same repository the first time a file beginning with the other ordered lists of header files specified in **HR.cf** is compiled with the +**k=/export/utils2/HR.cf** switch.

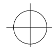




Avoiding recompiles

There are several restrictions relating to the use of precompiled header files that may result in files being recompiled when you expect them to be restored. For example, any of the following may result in a recompile: changing the order, number, or arguments of certain command-line switches such as -**I** and -**D**, unsynchronized system clocks, and adding comments to code. For more information refer to the precompiled header files entry in the *ObjectCenter Reference*.

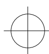


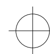


Part II: Getting started with your own code

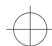
This part of the manual is designed to help you start using ObjectCenter with your own code. It shows you how to

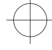
- *Load your own application into ObjectCenter*
- *Use ObjectCenter's run-time error checking, debugging, prototyping, and testing features*
- *Set options and customize ObjectCenter*

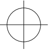


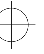


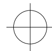
All these topics are discussed in more detail in the ObjectCenter User's Guide.

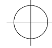










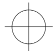


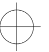
Chapter 6 Loading your application

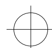
Before you begin debugging your own code, you need to set some load options, decide how to load your application, and then load your code and link in library modules.

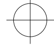
This chapter describes:

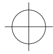
- *Setting load options*
- *Loading your application with the load command*
- *Loading your application with CenterLine targets*
- *Loading your application with the EZSTART utility*
- *Saving a Project file*
- *Troubleshooting the load and link commands*

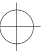


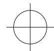


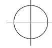












Getting ready to load your code

Before you can use ObjectCenter for debugging and visualizing your code, you need to load your application, and then link in library modules. You can load your application using three different methods:

- If your application does not use makefiles or is very small, load files manually with the **load** command (page 65).
- If your application uses a simple makefile, add CenterLine targets to it and use the ObjectCenter **make** command to load your application (page 66).
- If your application uses complex makefiles, use the EZSTART utility to create a new makefile with CenterLine targets, and then use the ObjectCenter **make** command to build the targets (page 70).

Performance issues

As mentioned in 'Trade-offs' on page 16 and 'Loading the Bounce program' on page 63, whether you use the **load** command or the **make** command, you can also choose to load your application in several different forms. For maximum error-checking and debugging, load it as source code. For maximum speed, load it as an executable in process debugging mode.

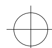
You can achieve a balance between speed and debugging capabilities by using a combination of source code and object code with and without instrumentation and debugging symbols. Please refer to the **performance** entry in the *Reference* for more information.

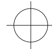
ObjectCenter provides two special features to improve performance: **demand-driven code generation** and the **precompiled header files** facility. Refer to the *Reference* for more information. You may also find 'Speeding up compilation with precompiled headers' on page 53 helpful.

Using options to set load switches

Before loading your application, you may need to set the **sys_load_cflags**, **sys_load_cxxflags**, and **load_flags** options to control the switches that are passed to the **load** command.

The **sys_load_cflags** and **sys_load_cxxflags** options specify site-wide switches. The **load_flags** option specifies default switches passed to the **load** command if the load command is issued *without any switches* apart from -**l** and -**C**.





Setting system-wide load options

ObjectCenter presets the **sys_load_cflags** option to search for C files and **sys_load_cxxflags** to search for C++ files. ObjectCenter always appends the contents of these options to all **load** commands.

The value of the system-wide options varies with each platform. To verify that your **sys_load_cflags** and **sys_load_cxxflags** options contain the correct values for your platform, enter these commands in the Workspace:

```
C++ 10 -> printopt sys_load_cxxflags
C++ 11 -> printopt sys_load_cflags
```

Normally, you use the default value in your system. If you have a different library or header file path from that specified by the **sys_load_cflags** and **sys_load_cxxflags** options, change the value of the option using the **setopt** command:

```
C++ 10 -> setopt sys_load_cxxflags switches
C++ 11 -> setopt sys_load_cflags switches
```

Instead of changing the value of an option, you can also add additional switches to the value of an option using the **#$** syntax. For example, the following commands add -**DBeta** to the list of switches:

```
C++ 10 -> setopt sys_load_cxxflags #$sys_load_cxxflags -DBeta
C++ 11 -> setopt sys_load_cflags #$sys_load_cflags -DBeta
```

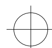
Setting the load_flags option

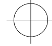
You can use the **load_flags** option to designate any switches specific to your own work. For example, a macro used in your project could be entered as follows:

```
C++ 1 -> setopt load_flags -DBETA -DDEBUG
C++ 2 -> load xyz.C
Loading (C++): -DBETA -DDEBUG -L/usr/include xyz.C
```

In this example, the -**L/usr/include** switch was defined in **sys_load_cxxflags** and automatically appended to the switches specified by the **load_flags** option. The **load_flags** option is automatically appended to the **load** command if you specify only the -**l** or -**C** switches on the command line. If you specify other switches on the command line, the **load_flags** option is ignored.

The method you choose for loading your application determines whether or not you need to set **load_flags** now. If you don't have a makefile with your application, set the **load_flags** option to contain all switches that apply globally to all the files in your application. If you're using EZSTART or CL targets to load your application, don't set **load_flags** now.





Loading your application with the load command

If your application doesn't use makefiles, then use the **load** command directly in the Workspace to load your application. For more information on syntax, refer to the **load** entry in the Manual Browser. To load your application:

1. Load the files to be debugged as regular object files with debugging information:

   ```
   C++ 21 -> load switches filename.o...
   ```

   Make sure you load any C modules with the -**C** switch.

2. Load the rest of the files that constitute your application as regular object files without debugging information:

   ```
   C++ 21 -> load -G switches filename.o...
   ```

3. Link any attached libraries with the **link** command:

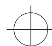
   ```
   C++ 21 -> link
   ```

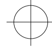
The following is an example of loading your application without a makefile. In the example, the **abc.o** module is the module being debugged, so it's loaded with debugging symbols.

```
C++ 1 -> ls *.o
abc.o xyz.o
C++ 2 -> setopt load_flags -DDEBUG
C++ 3 -> load -lm -C abc.o
Attaching: /usr/lib/libm.so
Loading: abc.o
C++ 4 -> load -G -C xyz.o
Loading: xyz.o
C++ 5 -> link
```

In this example, the **setopt load_flags** command precedes the **load** command so that the -**DDEBUG** switch is automatically appended to any **load** command that is issued without any switches or with only the -**C** and/or -**l** switch.

The -**DDEBUG** switch doesn't appear in the "Loading:" line because it will only be used if the **abc.o** file needs to be recompiled directly by ObjectCenter (without using a makefile) or if the file is later swapped to source using the **swap** command. It will *not* be used if the **xyz.o** file is recompiled because the **load_flags** option is ignored when the -**G** switch is used on the **load** command line.





Loading your application with CenterLine makefile targets

If your application uses one or more UNIX makefiles, you can add a CenterLine (CL) target that automatically uses the **load** command to load your application. CL targets are designed to work with the ObjectCenter **make** command.

NOTE If you have a complex makefile, refer to the **make** entry in the Manual Browser before designing your CL target.

To load your application with CL targets, you take these basic steps:

1. Design a CL target that automatically loads and links the object files in your application. The files to be debugged should be loaded as regular object code *with* debugging information, and the rest of the files as regular object code *without* debugging information.
2. Issue the **make** command from the ObjectCenter Workspace to execute the CL target.

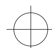
Designing CL targets

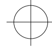
A CL target consists of the following lines:

- A *dependency line*, which specifies the target's dependencies
- *Shell lines*, which specify UNIX shell commands to be executed
- *CL lines*, which specify Workspace commands to be executed

The example shows a sample makefile that includes a CL target. Refer to this sample as you create your own CL targets. The sample makefile has the following three targets. The first two are standard **C**++ targets.

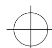
- The first target (**.C.o**) is an implicit target that specifies how to convert a **.C** file (C++ file) to a **.o** file (an object file). In this case, **CC** is called with the switches +**d**, -**g**, and -**c**.
- The second target (**all**) creates an executable named **my_program** from the three files **a.o**, **b.o**, and **c.o** which are specified in **$(OBJS)**. If any of the **.o** files are missing or out-of-date, they will be compiled, using the implicit target **.C.o**.

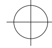




- The third target (**ocenter_obj**), a CL target, loads the objects specified in **$(OBJS)** (**a.o**, **b.o**, and **c.o**) into ObjectCenter with the switches specified in **$(CXXFLAGS)**.

```
# This is a standard makefile comment
# a.C, b.C, and c.C are C++ files

CC = CC
SRCS = a.c b.c c.C
OBJS = a.o b.o c.o
CXXFLAGS = -g -dd=off-DDEBUG
.SUFFIXES: .C .o

# Standard targets
# The following  is an implicit target that specifies
# how to convert a .C file to a .o file
.C.o:
   CC +d -g -c $<

all: $(OBJS)
   $(CC) +d $(CXXFLAGS) -o my_program $(OBJS)



# CL target
# Note the indented # character in CL lines

# The following target loads object files into
# ObjectCenter, using the implicit target
# to convert .C to .o
# The first line is the dependency line
# The second line is a shell line
# The remaining lines are CL lines
ocenter_obj:
      echo "Starting a CL obj target"
      #setopt program_name my_program
      #load $(CXXFLAGS) $(OBJS)
      #link
```



To create your CL target:

1. Edit your makefile.
2. Create a dependency line for the CL target.

   As shown in the sample makefile, the dependency line for a CL target follows the same syntax as that for a standard target. The ObjectCenter **load** command automatically follows the implicit suffix rules of the standard UNIX **make** utility for **.o** and **.C** files. Thus, in this example, the dependencies don't need to be specified.
3. (Optional) Create shell lines for the CL target.

   A shell line consists of a Tab followed by any number of shell commands, separated by semicolons:

   ```
   <TAB><shell command [; shell command ...] >
   ```

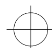
   From the perspective of the ObjectCenter **make** command, a shell line in a CL target is the same as entering the **sh** Workspace command with the shell line as an argument.
4. Create CL lines for setting any necessary Workspace options, such as **program_name**. (See 'Setting up your environment' on page 99 for more information about options.)

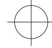
NOTE If you set the **load_flags** option in a makefile rule then it will affect all the **load** commands that you issue in the Workspace.

   A CL line begins with a Tab followed by # and any single Workspace command:

   ```
   <TAB>#<ObjectCenter command>
   ```

   Before ObjectCenter executes a rule that begins with a #, it passes the rule through the Bourne shell, just as **make** does. Using ## on a CL line prevents metacharacter expansion by the Bourne shell. Any line with a # character in the first position (without the Tab) is treated as a comment. A CL line is equivalent to entering the ObjectCenter command in the Workspace.
5. Create CL lines for loading the object files that make up your application. If necessary, refer to the **load** entry in the Manual Browser for information on syntax and switches. You should





load most files as regular object files *without* debugging information (**load -G**) and load the remainder (the modules you plan to debug) as regular object files *with* debugging information. Any C modules should be loaded with the **-C** switch.

If desired, you can use macros, such as **$(OBJS)**, as arguments to the **load** command. If the object files are out of date, then ObjectCenter invokes **make** to rebuild the files before loading.

In the example on page 67, note that the same set of switches used in the standard target, **$(CXXFLAGS)**, is also included in the **load** command for the **ocenter_obj** target. In this example, **$(CXXFLAGS)** contains the **-g** option, which means that files are compiled with debugging information. To ignore the debugging information, you need the **-G** switch with the **load** command.

6 Create a CL line for linking your application with any attached libraries:

```
<TAB>#link
```

Using the ObjectCenter make command

Once you have added your CL targets to your makefile, you can use the ObjectCenter **make** command on them. The ObjectCenter **make** command treats standard targets differently from CL targets.

When you issue the **make** Workspace command on a standard target, ObjectCenter invokes the UNIX **make** utility to build the target. ObjectCenter simply passes each command line to the Bourne shell for execution. The files are linked with the UNIX linker. The application is not loaded into ObjectCenter.

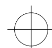
When you issue **make** on a CL target, the ObjectCenter commands specified in the target are executed. All the source files, libraries, and object files that are specified in the **load** command are loaded into the environment.

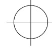
To load your application into the environment, invoke the **make** command on the CL targets that you added to your makefile:

```
C++ 20 -> make CL_target...
```

If you encounter problems

If you have problems designing your CL targets, refer to the make entry in the *ObjectCenter Reference*. If you have trouble using the **load** and **link** commands, refer to the **load** entry in the *ObjectCenter Reference* or 'Troubleshooting the load and link commands' on page 73.





Creating a makefile with the EZSTART utility

If your application has complex makefiles, consider using the EZSTART utility to create a makefile to load your application into ObjectCenter.

What is EZSTART?

EZSTART uses your existing makefile to generate a new makefile (**Makefile.cline**) containing appropriate CenterLine (CL) targets. Each CL target includes the appropriate commands for loading the files that constitute the corresponding make target. CL targets are designed to work with the ObjectCenter **make** command.

Leaving your original makefile unchanged, EZSTART monitors your existing makefile as it builds your current application once. Since EZSTART uses your existing makefile, it will only record information about commands that **make** executes. Based on this build of each file in your application, EZSTART constructs the makefile **Makefile.cline**, which contains the equivalent CL targets. Thus, you don't have to change your existing makefile to begin using ObjectCenter.

With **Makefile.cline**, you can use the ObjectCenter **make** command to load your files into the environment. Then you can set properties for the files, set various options, and save your session as a project file. As an alternative, you can customize the targets in **Makefile.cline**, integrate them into your existing makefile, and use the **make** command to establish your project at the beginning of each session.

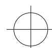
Using EZSTART

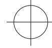
To use EZSTART as it is shipped, your application must:

- Use standard tools. By default, EZSTART recognizes only the following tools: **cc**, **clcc**, **CC**, **gcc**, **acc**, **ld**, **make**, **ar**, **mv**, and **cp**.
- Invoke these standard tools without using absolute pathnames. For example, your makefile should contain lines like **CC = cc** instead of **CC = /bin/cc**.

If your application does not meet these criteria the documents listed below may help you configure EZSTART to suit your application.

For usage details and troubleshooting information, please refer to the README, GETTING_STARTED, and REFERENCE_GUIDE documents in the **CenterLine**/*arch-os*/**EZ** directory, the **clezstart** manual page, and/or the "clezstart" and "I'm having trouble loading with make" entries in the Manual Browser.





Saving your project

By saving your project in a project file or maintaining its state in the CL targets in your makefile, you can quickly start up ObjectCenter with your application. This section describes how to save the state of your project in a project file or maintain the state of your project in the CL targets in your makefile.

Project files

A *project file* is a text script file that contains the information that ObjectCenter needs to rebuild your project across sessions. It records such information as:

- The files that make up the project and in which form they are loaded
- Warnings that have been suppressed
- The values of ObjectCenter options
- The signals that are caught and ignored
- The debugging items that have been set (such as breakpoints and actions)

A project file *doesn't* specify dynamic run-time information, such as variable values or break-level location, or information about your environment, such as the version of ObjectCenter you invoked or the type of workstation or terminal you are using.

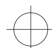
Saving your project file

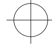
At any point, you can save the state of your project in a project file. During your ObjectCenter session, you can load the project file and quickly recapture the state of your work.

To save your project file, issue the **save** command:

```
C++ 56 -> save filename
```

ObjectCenter saves the script in *filename*. If you don't provide a value for *filename,* ObjectCenter saves the script in **ocenter.proj**.





Loading your project file

You can load a project file that you saved in several ways. You can:

- Specify the file when starting ObjectCenter:

```
% objectcenter ocenter.proj
```

- Load the project file from the Workspace:

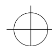
```
C++ 1 -> load ocenter.proj
```

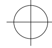
- Choose the **Load** menu item from the **File** menu in the Main Window or Project Browser and supply the name of the project file in the dialog box.

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

CL targets

A project file is not the only way for you to maintain a project in ObjectCenter. If desired, you can use the CL targets in your makefile to maintain your project. Since you can maintain your CL targets as other targets in your makefile change, you may find this model more natural for the long term and the project file model more practical for short-term tasks involving a subset of your application's modules.



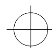


Troubleshooting the load and link commands

The following table gives you general information about finding and solving problems while you are loading and linking files. There is additional information in the *User's Guide* in the "Frequently asked questions" appendix and in the **load** and **link** entries in the *Reference*.

Table 6 Resolving **load** and **link** Problems

load or link Problem	Possible Solutions
The file specified with the **load** command is not found	Verify that the **path** option includes the directory that contains the source or object file specified with the **load** command.
Library not found	The **path** option does not search for header files or libraries. Verify that the -**I** and -**L** switches set the appropriate search paths for header files and libraries (respectively). The -**I** and -**L** switches can be passed with the **load** command or set with the **load_flags**, **sys_load_cflags**, and **sys_load_cxxflags** options. Set the -**I** and -**L** switches to appropriate values. ObjectCenter searches libraries in the order indicated in the **load** entry in the Manual Browser. You may need to unload the library and reload it for the **load** command to recognize the new flags.
Unresolved references after link	Use the **unres** command to get the list of unresolved references. Use the **link** -**list** command to check the library link order. Try issuing the **link** command a second time. Since ObjectCenter's linker makes only one pass, a second **link** command may resolve the references. Issue the **contents** command in the Workspace to verify you have the correct libraries loaded. Make sure that the correct -**I** and -**L** switches are being included in the **load** command by examining the **load** commands echoed to the Workspace. Make sure you load C files with the -**C** option. Modify your **load** switches or **load_flags** option appropriately.



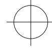


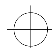
Table 6   Resolving **load** and **link** Problems (Continued)

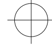
load or link Problem	Possible Solutions
	Explicitly unload files that have been modified or are affected by your changes to the **load_flags** option. To do so, issue the **unload** command: `C++ 1 -> unload file` Reload the files with the **load** command and relink the project with the **link** command.
Source module unloaded by ObjectCenter	This means an error exists in your source file. Check the Error Browser to understand the nature of the error. Make sure that you have included the right switches to locate header files by examining the **load** commands echoed to the Workspace. Invoke the editor if you need to correct a problem in your code. Explicitly unload the file in which the violations occurred by issuing the **unload** command in the Workspace. `C++ 1 -> unload file` Load the file again and relink your application.
Object module unloaded by ObjectCenter	If ObjectCenter has trouble reading the debugging information, you should load your file with the -**G** switch.
Cannot locate header file	Make sure that the correct -**I** flags are being included from the **load_flags**, **sys_load_cflags**, and **sys_load_cxxflags** options.
Cannot load library file	Make sure that the correct -**L** flags are being included from the **load_flags**, **sys_load_cflags**, and **sys_load_cxxflags** options.

TIP: Saving load-time error messages to a file

If your application generated a large number of warnings and errors in the Error Browser, you may find it useful to save them to a file. To do so, either choose **Write Messages to File** from the **Other** menu in the Error Browser, or enter the following command in the Workspace, where *source_file* is the name of the source file generating the errors, and *msg_file* is the name of the text file in which to save the errors:

```
C++ 37 -> load source_file #> msg_file
```

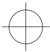



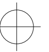


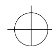
Chapter 7 Running your application

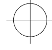
This chapter describes:

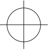
- *Running your program*
- *Understanding run-time error checking*
- *Instrumenting your application*
- *Responding to run-time problems*
- *Swapping a file from object to source*
- *Debugging techniques*
- *Rebuilding your project*
- *Exploring, enhancing, and testing your application*
- *Troubleshooting run-time issues*

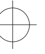


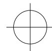


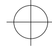












Running your program

Using the run command

When you've loaded your application into ObjectCenter, run your program by using the **run** command in the Workspace. The **run** command executes your **main()** function after initializing all variables and processing any command-line arguments.You can pass arguments to your program with the **run** command as follows:

```
C++ 2 -> run argument1 argument2
```

Naming your program

When a program is run from a shell, the value assigned to the variable **argv[0]** is the name of the program. In ObjectCenter, the value of **argv[0]** is set to the **program_name** option, for which the default value is **a.out**. You can change the value of **argv[0]** by setting the **program_name** option:

```
C++ 3 -> setopt program_name test
```

The Run Window

ObjectCenter starts as a background process, and a new **xterm** client, called **clxterm**, becomes the Run Window. The Run Window separates program input and output from the Workspace. When your application runs in ObjectCenter, the Run Window becomes the standard input and standard output of your program.The **clxterm** client provides all the same menus and features as an **xterm** client. Refer to the UNIX **xterm** manual page for additional information.

You can direct program output to the Workspace instead of the Run Window by setting the **win_io** option to false with the **unsetopt** command:

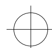
```
C++ 3 -> unsetopt win_io
```

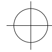
Detecting memory leaks

You can identify potential memomy leaks when you run your program by first setting the **mem_trace** option to the number of stack trace levels you want reported. For example, to generate a report with three stack trace levels, set the value of the **mem_trace** option to 3:

```
% setopt mem_trace 3
```

When you run your program, ObjectCenter creates a report showing the number of bytes in each potential leak, the size of the memory allocated, and the number of times each potential leak occurred. For more information, see the **memory leak detection** entry in the *Reference*.





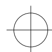
What is run-time error checking?

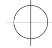
ObjectCenter's run-time error checking detects *dynamic violations*; that is, violations that occur during the execution of a program. These errors cannot be detected during compilation or with traditional programming tools or debuggers.

ObjectCenter does run-time error checking on both source and object code. Table 7 compares the kinds of run-time checking done for source files, instrumented object files, and object files (instrumented object files are explained on page 79).

Table 7 Types of Run-Time Error Checking

Type of File	Run-Time Warnings and Errors Issued
Source files	Memory allocation warnings Miscellaneous warnings (bad arguments to C library functions) Using memory that has not been set Addressing errors (pointer dereference, alignment, array index errors) Undefined/questionable arithmetic operations Undefined/illegal pointer operations Enumerator warnings Losing information during conversions/assignments Function warnings Storage warnings
Instrumented object files (with or without debugging symbols)	Memory allocation warnings Miscellaneous warnings (bad arguments to C library functions) Using memory that has not been set Addressing errors (pointer dereference, alignment, array index errors)
Regular object files (with or without debugging symbols)	Memory allocation warnings Miscellaneous warnings (bad arguments to C library functions)





The greatest amount of checking is done for source files (about 80 checks in all), ranging from undefined or illegal pointer operations to storage and function warnings.

The next greatest amount of checking is done for instrumented object files (about 10 checks in all), including checks on addressing errors and access of unset memory.

The least amount of checking is done for object files; you only get the run-time errors that occur in certain C library functions. ObjectCenter replaces many C library functions and system calls with its own versions of them, which contain enhanced error checking. See the C library functions section in the *Platform Guide* appendix to the online *Reference* for more information.

When ObjectCenter detects a run-time problem, it:

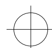
- Displays a message in the Error Browser.
- Updates the **Error Browser** button in the Main Window and Project Browser.
- Lists the corresponding source file in the Source area.
- Positions the Source area pointer at the line number causing the problem (for files loaded as source or object with debugging information).
- Generates a break level in the Workspace.

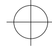
Adding more run-time error checking to object files

You can *instrument* both types of regular object code (with and without debugging symbols) to gain the advantages of run-time checking for pointer bounds errors and access to uninitialized memory.

Instrumented object code without debugging symbols runs slightly faster than instrumented object code with debugging symbols. ObjectCenter will detect errors in instrumented object code whether or not it has debugging symbols, but it will usually pinpoint the source of the error more exactly with debugging symbols.

You can instrument the files you loaded as regular object files with the **instrument** command, or with the **Instrument** button in the Project Browser.





Responding to run-time problems

Run-time *errors* are serious or potentially serious problems and should be fixed immediately. Run-time *warnings* are problems that are not serious enough to warrant an error. You can deal with run-time errors and warnings in three ways:

- Use ObjectCenter's environment to locate and correct the problem.
- Ignore minor warnings and keep them from appearing again.
- Decrease the amount of run-time error checking performed by ObjectCenter by setting a few options to filter out low-level warnings.

WARNING You can continue from a run-time error. However, continuing execution from a run-time error may create a non-recoverable internal error.

Using the environment to detect and correct a problem

If the error was detected in source code, use the debugging techniques described in 'Debugging techniques' on page 84 to debug your code.

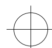
If the error was detected in object code, and you need more information to find the problem, you should do the following in ObjectCenter:

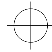
1. Swap the file from object code to source code using the **swap** command as described in 'Swapping a file from object to source' on page 82. This enables ObjectCenter to perform the most run-time error checking possible on your code.
2. Run your application again to allow ObjectCenter to locate the exact location of the problem.
3. Use the various debugging techniques described on page 84 to debug your code.

Ignoring the warnings

You can continue execution after receiving a warning by using the **Continue** menu item or by issuing the **cont** command from the Workspace.

In the Main Window, display the **Execute** menu and select **Continue**.





If you don't want to track certain warnings, you can tell ObjectCenter to ignore the warning condition from that point on. At any time, you can choose to suppress the reporting of particular warnings (but not errors). To suppress a warning:

1 Select the warning in the Error Browser.
2 Select one of the suppression scopes from the Suppress menu.

After you suppress a warning, ObjectCenter removes it from the Error Browser. You can, however, view suppressed warnings from the Suppressed Messages window by selecting **Open Browser** from the Suppress menu.

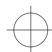
Dealing with large numbers of run-time warnings

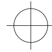
If you receive an overwhelming number of run-time warnings, you can set the following options to reduce the amount of run-time error checking that is performed on a file:

1 In the Main Window, display the **Browsers** menu and select **Options Browser**.
2 Display the **Option Sets** menu and select **memory**.
3 Set **save_memory** to **True**. With this setting, ObjectCenter does not report run-time warnings on dynamic type mismatches, dynamic used-before-set, or corrupted values, and it disables watchpoints on variables. ObjectCenter continues to check pointer bounds.
4 Set **unset_value** to **0**. With this setting, ObjectCenter does not report that a variable is used without being set, unless the value of the variable is 0.
5 Select the **Apply** button to apply the values.
6 Select the **Dismiss** button to close the Options Browser.

To turn these run-time checks back on, set **save_memory** to **False** and **unset_value** to **191**.

NOTE You can also suppress warnings on a per-file basis from the Project Browser. Select the file from the Files area, and then select the **Properties** button. In the Properties dialog box, select the **Ignore Warnings When Loading** check box and then the **Apply** button.





Swapping a file from object to source

You can use the **swap** command to swap between source and object code, replacing a source file with its object counterpart or an object file with its source counterpart.You can also select one or more object files from the Files area in the Project Browser and select the **Swap** button at the bottom of the Files area.

Typically, you swap a file from object to source when you need more extensive error checking on a file. When you've completed development and testing of a source file and want to improve load-time and run-time performance, swap the file from source to object.

Using the swap command

If the file you swap is loaded in object form, ObjectCenter unloads the object file and loads the corresponding source file. ObjectCenter determines whether to load the source file as a C++ or C file from the way in which you loaded the object file. If you used **-C** to load the object file, ObjectCenter loads the source as a C file; otherwise it loads the source as a C++ file, as shown in the following example:

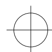
```
C++ 5 -> load a.o          /* C++ code */
Loading: a.o
C++ 6 -> swap a.C
Unloading: a.o
Loading (C++): a.C
C++ 7 -> load -C b.o       /* C code */
Loading: b.o
C++ 8 -> swap b.c
Unloading: b.o
Loading (C): b.c
```

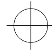
After swapping your object file to source, you should select the **Run** button (or issue the **run** command) so ObjectCenter can pinpoint the exact location of your run-time problem.

Setting the swap_uses_path option

When you swap an object file, ObjectCenterooks for the corresponding source file in the same directory in which the object file resides. If you have source and object files in different directories, you need to adjust the following options:

1 Set the **path** option to include the directories that contain the source files and object files.





2 Set the **swap_uses_path** option to **True**. When this option is set, ObjectCenter searches in the directories specified in the **path** option for the source file.

For example, if you set **path** to **/s3/beth/src:/s3/beth/obj** and **swap_uses_path** to **True**, and then swap a source file or an object file, you will see the following result:

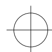
```
C++ 24 -> load test.C
Loading (C++): /s3/beth/src/test.C
C++ 25 -> swap test.C
Unloading: /s3/beth/src/test.C
Loading (C++): /s3/beth/obj/test.o
C++ 26 -> swap test.o
Unloading: /s3/beth/obj/test.o
Loading (C++): /s3/beth/src/test.C
```

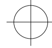
Troubleshooting the swap command

Table 8 contains information on possible solutions to swapping problems.

Table 8 Troubleshooting Swapping

Error Conditions	Possible Solutions
The **swap** command cannot find the source file or object file.	If you are swapping from object to source, determine whether the corresponding source file exists and include the correct directory in the **path** option. Set the **swap_uses_path** option.
The header files are not found when swapping from object to source.	Check the flags used to load the object file. ObjectCenter uses those flags to load the corresponding source files. Unload the object file and load it with the correct include switches. Issue the **swap** command. Remember that you have the option of loading the source file directly into ObjectCenter.
The right macros are not defined when swapping to a source file.	Unload the file using the **unload** command and load it again with the right flags before issuing the **swap** command.





Debugging techniques

ObjectCenter provides full interactive debugging for C++, including support for overloaded names and operators, classes, and virtual functions. In this section, we show you some debugging techniques that you can try on your application.

In addition to using the interpreter in the Workspace, you can use an extensive set of debugging commands to:

- Set breakpoints and watchpoints.
- Define actions to execute when particular lines of your code are reached during execution, or when particular variables are modified.
- Trace execution of your program.
- Step through your program.

Debugging commands

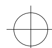
In component debugging mode, use the **next**, **step**, **stepout**, **cont**, **up**, and **down** commands to explore your application. In process debugging mode you can also use **stepi** and **nexti**. Consult the corresponding entries in the *Reference* to learn more about these basic debugging commands. Keep in mind that most of these commands can be used only on object files with debugging symbols or source files. If necessary, reload your files with debugging information

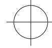
Using the Interactive Workspace

The Interactive Workspace provides you with the complete programming environment at a breakpoint, not just access to a small subset of commands. In addition to ObjectCenter commands, you can enter any legal source code in the Workspace. Source code entered in the Workspace should be terminated by a semicolon (;) to distinguish it from ObjectCenter commands. When you enter code into the Workspace, ObjectCenter parses your code like the C compiler or the C++ translator.

The following examples use the Bounce program:

- Use the Workspace to examine any variables within the current scope. In the following example, ObjectCenter detected a run-time error, stopped execution of the program, and generated a break level. You can examine the **count** variable.





ObjectCenter displays all the declarations of **count** and indicates which declarations are active.

```
C++ (break 1) 22 -> whatis count
int count; /* Formal of 'DrawableShape::drawMove;
currently active */
auto int count; /*Defined in 'DrawableShape::doDraw;
currently inactive */
```

- Use the Workspace to instantiate an object at a breakpoint. The following example defines a variable named **pt**.

```
C++ (break 1) 23 -> count;
(int) 400
C++ (break 1) 24 -> Point pt(400,536);
(class Point *) 0x2632a0 /* (class Point) pt */
```

- Invoke the member function **draw** at the breakpoint:

```
C++ (break 1) 25 -> draw();
```

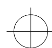
- Develop code fragments or functions using all your program variables and functions that are in scope. These code fragments can be designed to extensively debug and test your code. You can also set variables to any value at a breakpoint and then continue execution.

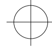
NOTE If the input prompt changes to a "+>" while you enter code fragments in the Workspace, the Workspace expects you to supply additional input. This often happens when you forget to type a semicolon (;) at the end of a C or C++ statement.

For a complete description of Workspace features, refer to the **Workspace** entry in the *Reference*.

Visualizing data structures at run time

While you are running your application, you can graphically monitor data with the Data Browser. It is particularly useful for viewing complex data structures with many levels of indirection through pointers. To display the Data Browser, select an identifier or expression in your application and use the **Display** menu item. The identifier must be a variable in scope or an expression. If the variable contains any pointer boxes, you can select them to dereference the pointers. To close the Data Browser, select the **Dismiss** button.





You can change the settings of the following options to display objects differently in the Data Browser:

- **print_inherited**
- **print_runtime_type**
- **print_pointer**
- **print_static**
- **show_inheritance**

Consult the **options** entry in the *Reference* for more information about these options.

Displaying run-time or compile-time type information

When displaying a reference or dereferencing a pointer in a running program, ObjectCenter displays the runtime type of the object being pointed to. If you'd rather display the definition-time ("compile-time") type than the run-time type, unset the Boolean option **print_runtime_type**, which is set by default.

```
C++ 7 -> unsetopt print_runtime_type
```

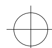
Examining classes and class members

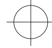
You can graphically examine the classes and class members in your loaded program with the Inheritance Browser. The Inheritance Browser is useful for understanding the complex relationships among classes. To display the Inheritance Browser and the Class Examiner:

1. Select the name of a class in your program in the Source area.
2. In the Main Window, display the **Examine** menu and select **Examine Class**.

The Inheritance Browser lists all loaded classes, and the Class Examiner lists all the members of the class you selected from the Source area. To display the relationships between classes, select items from the Class List in the Inheritance Browser.

You can print the contents of the Inheritance Browser (and 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.





Dynamically inserting statements in your code

You can use the **action** command or Action dialog box to customize and extend the built-in debugging facilities of ObjectCenter. The **action** command lets you insert C or C++ code or ObjectCenter commands into your application without actually changing your program.

Print the number of times a function was called

For example, you can use the **action** command to calculate the number of times a certain function or procedure was called. Try this example with a procedure in your application.

1. Define an integer in the Workspace to serve as the global data structure, and initialize it to zero.

```
C++ 12 -> int count;
C++ 13 -> count = 0;
```

2. In the Main Window, display the **Debug** menu and select **Set Action**.
3. In the **Function** text field, enter the name of the procedure on which to set the action.
4. In the **Action Body** text box, enter the following code:

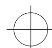
```
{
printf("Adding on to the count \n");
++count;
}
```

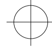
5. Select the **Set Action** button.
6. Rerun your application and watch the "Adding on to the count" message appear in the Run Window.

Automatically printing data structure values

You can set an action at any line in your code or on a function definition. By setting the action on a function definition, you can print data structure values or the arguments to a procedure. Remember, you can specify either valid C or C++ code or CenterLine commands as part of your action.

NOTE Setting actions on a function definition requires the file containing the function to be loaded as source or as object code with debugging information.





Setting conditional breakpoints

You can set conditional breakpoints using the **action** command or Action dialog box. Use this feature to generate a break level if certain conditions are true. Try the following example with your application to generate a break level if the variable **i** is greater than 3.

1. Open the Action dialog box. To do so, in the Main Window, display the **Debug** menu and select **Set Action**.
2. In the **Function** text field, enter the name of the procedure on which to set the action.
3. In the **Action Body** text box, enter the following code, except replace *i* with the name of a counter variable that is in scope in your application:

```
{
if (i > 3) (
        centerline_stop(" ")
        centerline_whereami(" ");}
printf("i = %d \n, i);
}
```

   All ObjectCenter Workspace commands, such as **stop**, have a C or C++ function equivalent that you can call from C or C++ code. The function equivalent for any command is the name of the command with the prefix **centerline_** added to it.

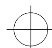
   All **centerline_** functions take one argument, a string. If the ObjectCenter command does not take any argument, you need to use an empty string as the argument when using the function equivalent.

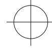
   In the example, **centerline_stop()** is the equivalent for the ObjectCenter command **stop**.

4. Select the **Set Action** button.
5. Rerun or continue the execution of your application to see the break level generated.

Viewing implicit function calls

When you look at C++ code, it can be difficult to determine precisely what C++ statements are called when a single C++ statement is executed. For example, a statement may cause overloaded operators and functions to be called or a user-defined conversion to be invoked. Object Center provides a powerful tool for disambiguating C++ code: the **expand** command.





With **expand**, you can see which functions can 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 thus be a subset of the functions listed by **expand**. The **expand** command evaluates the selected statements; it does not execute them.

You can use **expand** to:

- Understand the performance characteristics of a certain class. You can see exactly how the constructors and destructors are defined and make decisions to improve the performance of a class. For example, you can decide to make the constructors inline.
- Trace the execution of a function. You use **expand** to determine the first function call. Then you can set a breakpoint on that function and use either the **next** or **step** command to trace its execution.
- Understand all the member functions of a class.

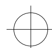
Getting information about addresses

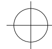
You can use the **info** command to display the name, size, and type of the object associated with an address. If the address refers to allocated data, **info** displays the size of the allocated data and, if available, the type of data most recently stored there. The **info** command also indicates if the address is being watched by a debugging action or contains a bad pointer.

Try the following example. Use **info** to display information about the address stored in the variable **ptr**, which points to the fourth element in the array **many**.

```
C++ 1 -> int *ptr;
C++ 2 -> int many[10];
C++ 3 -> ptr = &many[4];
(int *) 0x270790 /* many[4] */
C++ 4 -> info ptr
address = 0x146c28, name = ptr
Size = 4, contains type: pointer.
```

NOTE You can use the **info** command only with source files or object files with debugging symbols.





Monitoring a memory location

You can set a watchpoint, which interrupts execution whenever a specific address is modified. You can set watchpoints on the addresses of global variables, allocated data, formal parameters, and automatic variables.

When you use watchpoints, be aware of the following:

- The variables need to be in scope when you set the watchpoints.
- If the watched address is modified within object code, ObjectCenter does not detect the event, and execution of the program is not interrupted.

You use the **stop on** command to set a watchpoint. Try the following example to set a watchpoint on the variable **abc**:

```
C++ 1 -> int abc;
C++ 2 -> stop on abc
stop (1) set on address 0xd9616.
```

The number of bytes watched equals the size of the type of data. In the previous example, four bytes are watched because **abc** is an **int**, and the size of an **int** is four bytes.

To set a watchpoint on the address stored in a pointer, the argument to **stop** should be the value of the pointer. Try this example:

```
C++ 1 -> int *ptr;
C++ 2 -> ptr = new int[20];
(int *) 0x179d8c /* (allocated) */
C++ 3 -> stop on *ptr
stop (1) set on address 0x179d8c.
```

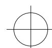
Debugging fully linked executables

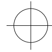
You can load a fully linked executable or corefile in process debugging mode. Working with an executable demands the least amount of memory and runs at the full speed of the machine. You can immediately locate an error that causes a crash by specifying a corefile and also use standard **gdb** commands for debugging.

Keep in mind, however, that in process debugging mode no load-time or run-time error checking is available, and the Inheritance, Cross-Reference, and Project Browsers are not available to aid you in visualizing functions and files in your application.

Start ObjectCenter in process debugging mode with the -**pdm** switch, and use the **debug** command to load your executable:

```
% objectcenter -pdm
(pdm) 1 -> debug a.out
```



For complete information on process debugging mode, refer to the pdm and **debug** entries in the *ObjectCenter Reference*.

Debugging multiple processes

ObjectCenter provides a unique environment to debug applications that have multiple processes. If your program calls **fork()**, the child process appears in a separate window and shares a Run window with the parent process. You can set breakpoints in the parent and child independently. However, in the child process, your code must be loaded as source to set breakpoints; the parent process can be loaded as source or object code to set breakpoints. For more information on debugging forked processes, refer to the **debugging** entry in the *Reference*.

Debugging threaded processes

In process debugging mode, ObjectCenter provides the **thread** and **threads** commands and a graphical Thread Browser. The Thread Browser provides you with information about the threads and lightweight processes in your program, including a list of all threads, and the state of each thread.

For more information about debugging threaded applications, see the thread, threads, and thread support entries in the *ObjectCenter Reference*. Support for debugging threaded applications is not available on all platforms. See the “Product limitations” section in the online *ObjectCenter Reference* for details.

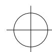
Using Ascii ObjectCenter

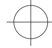
You can use Ascii ObjectCenter instead of the graphical version for any of the following reasons:

- To run ObjectCenter on a nongraphical workstation or over a dialup line.
- To gain faster startup time or to reduce the amount of memory needed to run ObjectCenter.
- To debug GUI programs. By using an ASCII terminal running alongside the X server, you can more easily debug programs that grab mouse and keyboard I/O.
- To do automated test runs. You can automate program tests by using I/O redirection with the **run** command and setting the **batch_load** option. See the **run** entry in the *Reference* for more information.

Start Ascii ObjectCenter with the -**ascii** switch:

```
% objectcenter -ascii
```



Rebuilding your project

After you try debugging techniques on your application and edit some of the files that are currently loaded, you need to update your project.

To reload all modified files in your project, build your project. You can build your project with the **build** command in the Workspace, the **Build** menu item on the **Session** menu, or the **Build** button in the Button panel. ObjectCenter reloads any files that are out of date.

The **build** command also attempts to reload any files that failed to load previously because they contained an error (files listed in the Project Browser as **failed**). The **build** command attempts to reload such files each time it is issued until it successfully loads the file or until you explicitly unload the file using the **unload** command.

Reloading source files

A source file is reloaded if the source file itself or any of the header files it includes have been modified since the file was loaded.

Reloading object files

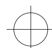
An object file that is older than its source counterpart is recompiled, then reloaded. Also, an object file is reloaded if the file has been recompiled since it was loaded. If there is a makefile in the directory containing the source file, ObjectCenter issues a **make** command. If there is no makefile in the directory containing the source file, ObjectCenter directly invokes the C++ translator or C compiler, whichever is appropriate.

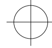
If the object file is loaded with debugging information, ObjectCenter checks header files that the object file depends on; the object file is recompiled and reloaded if it is older than any of the header files.

If you modified an individual file, you can reload it using the **load** command and specifying the name of the file.

Incremental linking

When ObjectCenter reloads files, it only relinks the files that have changed, which significantly reduces the link time. Relinking typically takes 2 to 10 seconds, depending on the size and type of the file or files that you modified. Incremental linking time is independent of application size; the more files in your application, the more time incremental linking saves.





Exploring, enhancing, and testing your application

While ObjectCenter provides you a complete environment for debugging, its powerful development environment works equally well for exploring, enhancing, and testing your existing application and developing new applications. You can use ObjectCenter for prototyping new applications and as a test harness.

Using the Workspace for interactive prototyping

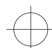
In component debugging mode, in addition to handling ObjectCenter commands, the Workspace functions as a direct interface to ObjectCenter's interpreter. Because the interpreter implements the full C++ language and the full C language as defined by Kernighan and Ritchie (K&R) and also offers support for the ANSI C standard, you can enter any statement in the Workspace and ObjectCenter executes it immediately. Also, when you execute code from the Workspace, ObjectCenter's run-time error checker automatically checks it for dynamic problems.

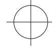
The default for the interpreter (K&R C or ANSI C) depends on the underlying compiler you are using, which is different for each platform. See the *Platform Guide* appendix to the online *Reference* for more information. For more information on using ANSI C code, see the ANSI C, code generation, and **config_parser** entries in the *Reference*.

Because the Workspace allows you to enter any C or C++ statement and immediately execute it, the Workspace supports component level development through interactive prototyping. You can define new C++ classes or redefine existing classes. You can create object instances on the fly by typing the corresponding C++ declaration or using the **new** operator. For example, to define a function that adds two integers, you could enter:

```
C++ 29 -> int add(int x, int y)
C++ 30 +> {return x+y;};
C++ 31 -> add(3,4);
(int) 7
```

While this example is extremely simple, you can easily extend this same approach to explore something more complex, such as a new string comparison routine based on the Boyer-Mohr algorithm. You would use the Workspace to interactively try out stepwise refinements of your algorithm.





Saving prototyped code with the edit workspace command

You can save code you define in the Workspace with the **edit workspace** command. During a session, all C and C++ definitions you enter are stored in a Workspace scratchpad. The **edit workspace** command lets you save the scratchpad to a file, by default **workspace.C**, and then edit the file.

For example, suppose you create a class in the Workspace. You can 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, you can use **edit workspace** to create a file containing the code you defined in the Workspace. You can enter a name for the file or accept the default.

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

Requesting edit of file '/net/my_proj/workspace.C',
line 1 ...
->
```

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. The **source** command will report errors if you've unloaded any definitions that the saved file depends on.

If you want to use the new file as source code, add any **#include** lines you need and remove any extraneous lines. 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.

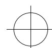
You can also save all your inputs in the Workspace at any point by redirecting the output of the **history** command:

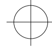
```
C++ 30 -> history #> ocenter_log_name
```

The **source** command reads and executes files containing any legal command that can be typed in the Workspace.

```
C++ 31 -> source command_file
```

The file *command_file* must contain valid Workspace commands and C and C++ code to execute in the Workspace.





Dual language support

One of the keys to the success of the C++ language is that it combines full support for object-oriented programming with the efficiency required for delivery of commercial applications and still retains compatibility with its predecessor, the C language. By default, ObjectCenter starts in C++ mode. That is, programming statements you enter in the Workspace are considered C++ statements and parsed accordingly.

To switch to C mode, use the **cmode** command:

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

To return to C++ mode, use the **cxxmode** command:

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

Unit testing

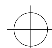
Constructing a modular, maintainable application is easier if you can extract a component from its context in the program, test and refine it in isolation, and then merge modifications back into the program. ObjectCenter‘s interpreter enables you to take this approach.

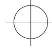
Using your editor and ObjectCenter‘s incremental loader/linker, you can create and load a small-to-medium functional unit or a group of functional units. Using the Workspace, you can test these units individually, as a set, and in interaction with your established project components. Once you have tested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

Loading incomplete programs into ObjectCenter

You can 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 have the choice of ignoring the unresolved references and executing only the section of code that you want to test.





Invoking individual functions from your program

You can load a complete program intoObjectCenter and invoke individual functions from the Workspace, rather than calling the function **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 have 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 the application.

System-level testing

ObjectCenter provides a powerful environment to test your complete application.

Running test suites

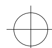
A typical development effort requires programmers to run a test suite on code that they modify before checking the code back into their version control system. You can automate this process by using the **source** command to run a test suite before you check your file into your version control system.

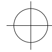
Executing your program with different test data

You may want to maintain the current state of your program when using different test data. When you use **run** or **rerun** to execute **main()**, all global variables are initialized because both commands call the **reinit** command before executing your program. If you use the **start** command instead, it performs all the functions of **run** without initializing global variables.

Try the following example. Set the variable **seed** to a special value before executing **main()**. To avoid having the value of **seed** reset to zero, use the **start** command instead of **run**. In the example **reinit** is called before **start**. The **reinit** command must be called between calls to **start** to ensure that input/output buffers and other library data structures are initialized to their correct values.

```
C++ 9 -> int seed;
C++ 10 -> seed = 7;
C++ 11 -> whatis seed
extern int seed; /* initialized */
C++ 12 -> seed;
(int 7)
C++ 13 -> reinit
C++ 14 -> seed;
(int) 0
C++ 15 -> seed = 7;
(int) 7
C++ 16 -> start
Executing: a.out
```

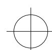


Troubleshooting run-time issues

The following table provides information about finding and solving problems while you are running, debugging, and enhancing your application. In addition to the information in the table, consult the “Frequently asked questions” appendix to the online *User’s Guide*.

Table 9 Troubleshooting Run-Time Issues

Problem and Issues	Possible Solutions
Application working outside environment does not work inside the environment	If you have run-time errors and warnings in your program, please refer to 'Responding to run-time problems' on page 80 to resolve run-time issues. If you have a working application and you just want to get your application working in ObjectCenter, you should refer to the information in 'Dealing with large numbers of run-time warnings' on page 81 for ways you can scale down the amount of run-time error checking performed by ObjectCenter. If your application uses the name of the application during execution, you must set the **program_name** option.
Not enough virtual space	If you experience slow response time or software crashes, you may not have enough virtual space. Check your swap space by issuing the appropriate command for your operating system in your shell, for example **pstat -s** or **/etc/swapinfo**. If you do not have enough swap space and you cannot increase it, try to reduce your requirements, for example by loading more of your files as object files without debugging information (**load -G**) rather than as source files.
Cannot instantiate class objects or call their member functions	Verify that the class definition has been loaded by browsing the class information with the Inheritance Browser. If the class definition has not been loaded in source, load the class definition by swapping an object file that contains the class definition to source, or directly load the header file containing the class definition in the Workspace with the **load_header** command.



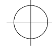


Table 9 Troubleshooting Run-Time Issues

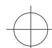
Problem and Issues	Possible Solutions
	Make sure that the file containing the implementation of the constructors and destructors is loaded with the -**g** switch and that it is not loaded with the +**d** switch. The +**d** switch suppresses the expansion of functions that are defined as inline.
Unresolved references unrelated to the functions you are calling in the Workspace.	If you call a function in the Workspace, all static constructors (**sti** routines) are executed, not just the static constructors for the file in which the function is defined. Either issue the **continue** command or resolve the references.
External preprocessors	ObjectCenter supports working directly with input files that are run through preprocessors that generate C++ or C files with the **#line** directives pointing back to the input file. Such preprocessors include YACC, certain SQL preprocessors, and preprocessors supporting parameterized types. See the **preprocessed code** entry in the *Reference* for complete information on this topic.
Slow performance during browsing of classes	Unset the **print_inherited** and **print_static** options. Unsetting these options reduces the amount of information displayed in the Inheritance Browser and Workspace.

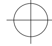
Cannot declare functions in the Workspace	If you declare a function in the Workspace, you must include the return type as part of the declaration. For example:
	
	`C++ 6 -> void func1() {`
	`C++ 7 +> printf("Hello world\n");`
	`C++ 8 +> }`

TIP: Exporting the contents of the Project Browser to a text file

If desired, you can dump the contents of the Project Browser to a file with the following Workspace command:

```
C++ 7 -> contents #> filename
```

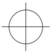


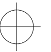


Chapter 8 Setting up your environment

ObjectCenter provides a number of options for tailoring the environment to suit your needs. There are also several other ways you can customize your environment.

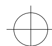
This chapter guides you through the basic set of options required to set up your environment and touches on other customizations, such as user-defined buttons, X resources, revision control, and editor support.

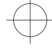
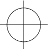
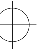
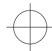


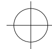


This chapter describes:

- *Setting options in the Workspace*
- *Saving your option settings*
- *Customizing your environment*





Setting options in the Workspace

Many Workspace commands can be controlled by setting the values of options. You can set options using:

- The Options Browser
- The Project-wide Properties, File Properties, and Library Properties windows in the Project Browser
- Workspace commands

This section shows how to set options with the Options Browser. If you need additional information on options, refer to the options , printopt , setopt , and unsetopt entries in the *Reference*.

The Options Browser and most options are available only in component debugging mode.

To open the Options Browser from the Main Window, display the **Browsers** menu and select **Options Browser**.

Setting the path option

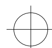
When ObjectCenter searches for files to load, list, edit, or swap, it looks in the directories specified by the **path** option. If the **path** option is not set, ObjectCenter searches only the current working directory.

NOTE The **path** option affects the loading, editing, and listing of source and object files only. It does not affect the loading of libraries and header files.

When setting the **path** option, you can specify absolute or relative pathnames for the directories. ObjectCenter searches the directories in the order that you specified and appends the current directory to the end of the list of directories. That is, if ObjectCenter doesn't find a file in the specified path, then it looks in the current directory.

In the following example, the **path** option is set in the Workspace so the **load** command looks for files in the **/usr/prog/test** directory before it examines the current directory.

```
C++ 1 -> setopt path /usr/prog/test
C++ 2 -> load abc.C def.o
Loading (C++): /usr/prog/test/abc.C
Loading: def.o
```

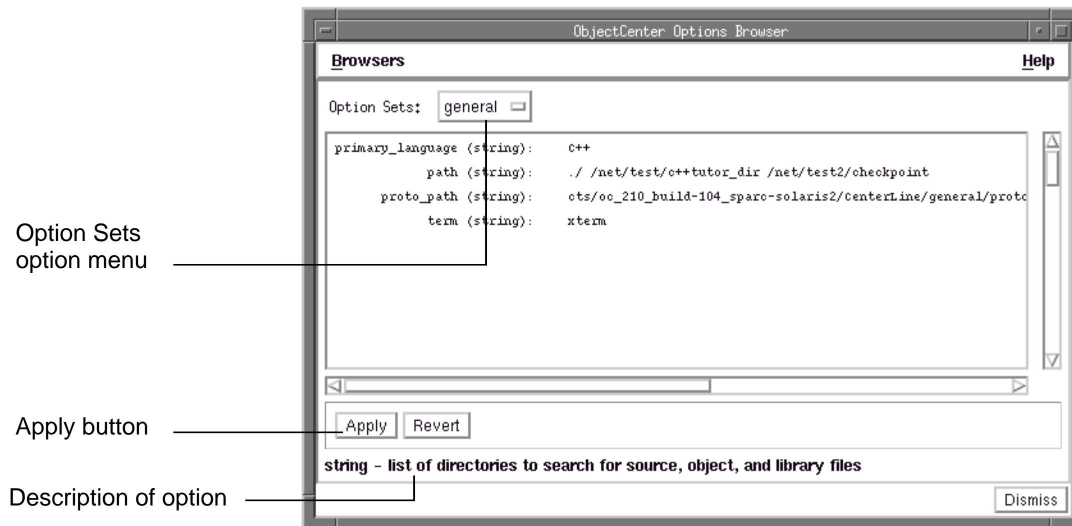



To set the **path** option from the Options Browser:

1 Display the **Option Sets** option menu and select **general**.

2 Select the text field next to the **path** option.

3 Enter the directories that contain all of your application's source and object files.



Applying your changes

To apply changes you've made in the Options Browser:

1 Select the **Apply** button to apply your changes.

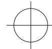
2 Select the **Dismiss** button to close the Options Browser.

Setting other options

Before you load your code, you may want to set the **sys_load_cxxflags**, **sys_load_cflags**, and **load_flags** options as described on page 63. Table 10 outlines additional options that you may need to set depending on your application.

Table 10 Additional Workspace Options

If your application...	Set this option...	For this result...
Uses ANSI C.	In the **misc** Option Set, set **ansi** to **True**.	Performs preprocessing and function prototype conversion in strict conformance with the ANSI C standard.
Contains object modules compiled with Sun C++ 3.0.1.	In the load Option Set, set **backend_ansi** to True	Causes ObjectCenter to produce ANSI C intermediate code.
Does not use **a.out** as the name of its executable and uses the executable name during execution.	In the **run-time** Option Set, set **program_name** to the name of your executable.	Uses the name specified as the value of the first argument, **argv[0]**, to **main()**.
Relies on a preprocessor, such as **m4** for macro expansion or an SQL processor for database processing.	In the **load** Option Set, set **preprocessor** to the command string to be executed in a subshell. The command string must have a **%s** in it, which is replaced by the name of the file being loaded, for example **m4 macro_file %s**	Executes the specified command in a subshell before loading the file. For complete information on using preprocessors, refer to the preprocessed code entry in the *Reference*.
Uses 8-bit character sets.	In the **misc** Option Set, set **eight_bit** to **True**.	Treats input and output as 8-bit characters.
Uses C++ source files with suffixes other than **.C** or **.c**.	In the **misc** Option Set, set **cxx_suffixes** to the list of filename suffixes for C++ source files, for example: **cpp cxx C c**	Searches for C++ source files using the file suffixes in the order specified. If the search fails, searches for a file with a **.C** suffix and then with a **.c** suffix.



Saving your option settings

Changes you make to ObjectCenter options are *not* saved automatically across sessions. There are two ways you can save your individual option settings across ObjectCenter sessions:

- Specify them in your ObjectCenter startup file.
- Save a project file and reload it in the next session. See the section 'Saving your project' on page 71 for more information.

Startup files in component debugging mode

When you start ObjectCenter in component debugging mode, it looks for the global startup file, **CenterLine/configs/ocenterinit**. It then searches for a local **.ocenterinit** startup file, first in the current working directory, and then in your home directory. If ObjectCenter finds the startup file, it executes all the commands in the file.

The **.ocenterinit** file is a text file that can contain any input that is accepted inthe ObjectCenter Workspace, including Workspace commands and source code that does not need to be debugged or reloaded. You can use the **.ocenterinit** file to store your option settings and aliases across sessions.

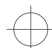
Here is a sample **.ocenterinit** file that sets two aliases (**s** for **step** and **n** for **next**), the **path** option, and the **tab_stop** option (number of spaces for tab expansion):

```
/* Define aliases for common commands. */
alias s         step
alias n         next

/* Specify option settings. */
setopt path ../test ../src
setopt tab_stop 4
```

NOTE 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.

Startup files in process debugging mode

In process debugging mode, ObjectCenter searches for a local **.pdminit** startup file, first in the current working directory, and then in your home directory. It does not read the global **ocenterinit** file.



Customizing your environment

You can create your own custom commands, which are placed on the **User Defined** submenu in the Main Window, and your own buttons to accompany your custom commands. You can also integrate your revision control system with ObjectCenter by using X resources, custom buttons, or the Workspace.

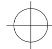
Creating custom commands

To create a custom command:

1. Display the **ObjectCenter** menu, slide off on the **User Defined** submenu, and select **Add/Change/Delete**. ObjectCenter displays the User Defined dialog box.
2. Enter the name of the command in the **Label** text field. This is the name that will appear on the menu item (and on a button, if you define one).
3. Choose either the **Workspace** or **Shell** radio button as the type of command.
4. Select the **Create Button** check box if you want to add a button to the control panel with the same name and function as the menu item.
5. If you chose **Shell**, specify the shell to be forked in the **Terminal** text field. You also can select either the **Wait for Completion** (wait for all shell commands to terminate before continuing) or **Run in Terminal Emulator** (direct shell output to terminal emulator) check box.
6. Type the command you want to create in the **Command Text** box. To see a list of variables you can use, place your pointer over the Command Text field and press F1, or refer to the **user-defined commands** entry in the Manual Browser.
7. Select the **Add** button. When you have finished entering commands, select the **Cancel** button.

ObjectCenter saves the commands you create across sessions. The information is stored in the file **.octrusrcmd** in your home directory.

To issue the custom command, simply choose the menu item or button.



Setting X resources

ObjectCenter provides a full set of X resources for tailoring the appearance and behavior of the GUI. For example, you can change colors and fonts, define special keyboard bindings, and even add custom buttons to integrate other tools (such as source control systems) into ObjectCenter.

For complete information on X resources, refer to the X resources entry in the *Reference*.

Revision control system support

If you use either the **rcs** or **sccs** revision control system, you can add **CheckIn**, **CheckOut**, **FileHistory**, and **FileDiffs** items to the **User Defined** menu in the Project Browser by setting the value of the following resource to either **rcs** or **sccs**:

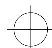
```
ObjectCenter*ProjectBrowser.RevisionControl:
```

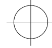
For more information about revision control system support, refer to the "Revision control systems" section in the *ObjectCenter Reference*.

Integrating revision control by creating C functions

ObjectCenter provides X resources for integrating your version control system into the Project Browser. As an alternative or an addition to using the built-in menu items, you can integrate your version control system into the Workspace by creating C functions that call both system and CenterLine functions. For example, consider the file **vc.c**:

```
#include <stdio.h>
checkout_and_load(char* arg)
{
char arrayco[200];
strcpy(arrayco, "co -l ");
strcat(arrayco,arg);
system(arrayco);
centerline_load(arg);
printf("Checkout and load are done\n");
};

checkin_and_swap(char* arg)
{
char arrayci[200];
strcpy(arrayci, "ci -l ");
strcat(arrayci,arg);
system(arrayci);
centerline_swap(arg);
printf("Checkin and swap are done\n");
};
```



You can load **vc.c** directly in the Workspace and then use its functions as follows:

```
C++1 -> alias co checkout_and_load("#$1")
C++2 -> load -C vc.c
Loading (C): vc.cc
C++3 -> co myfile.c
Loading (C++): myfile.c
Checkout and load are done
C++4 -> checkin_and_swap("myfile.c");
Unloading: myfile.c
Loading: myfile.o
Checkin and swap are done
C++5 ->
```

The alias **co** invokes the function **checkout_and_load**. The function **checkout_and_load** automatically checks a file out of the version control system and loads it into ObjectCenter in source form. The function **checkin_and_swap** checks the file back into the version control system and swaps it into object form in the environment.

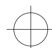
Using your own editor

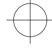
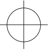
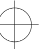
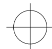
ObjectCenter supports **vi** and FSF GNU Emacs. The default editor is **vi**. To specify FSF GNU Emacs as your editor, use the following shell command:

```
% setenv EDITOR emacs
```

FSF GNU Emacs and **vi** are the *only* editors we support. However, the **CenterLine/API** directory contains a sample edit server for other editors, and unsupported editors for some platforms are in the **CenterLine/unsupported** directory.

You can also start ObjectCenter under the control of FSF GNU Emacs. Refer to the emacs integration entry in the *ObjectCenter Reference*.



Index

Symbols

(pound sign), in CL targets 68
+> prompt 85

A

actions, saving 71
addresses, getting information on 89
ANSI C
    additional documentation on 93
    intermediate code, generating 103
    setting options for 103
Ascii ObjectCenter, using 91

B

backend_ansi option 103
batch_load option 91
break levels
    establishing 79
    generating 88
breakpoints
    conditional 88
    saving 71
    setting in parent and child 91
build command 92
building, your project 92
buttons, creating 105

C

C compilers, ANSI option 103
C files, loading 50, 82
C functions, creating 106
C interpreter, using 93
C library functions, and run-time error checking 79
-**C** switch, for loading files 50
CenterLine targets, *See* CL targets
centerline_* functions 88
child processes, setting breakpoints 91
CL targets
    additional documentation on 66
    and EZSTART 70
    building 69
    designing 66
    for maintaining projects 72
    loading your application with 66 to 69
Class Examiner 22
    components of 86
class members, examining 86
classes
    examining 86
    performance characteristics 89
clxterm client 77
cmode command 95
code fragments, entering in the Workspace 85
commands
    build 92
    cmode 95
    cont 80
    cxxmode 95
    expand 89
    for setting options 101
    history 94
    in CL targets 66
    info 89
    make 69, 70
    run 77
    save 71
    setopt 64
    sh 68
    source 94
    start 96
    stop on 90
    swap 82
    unload 92
    unres 73

conditional breakpoints 88
constructors, examining 89
continuing, execution 80
creating
    buttons 105
    CL targets 68
    functions 106
Cross-Reference Browser 24
    printing contents 86
customizing your environment 104
cxx_suffixes option 103
cxxmode command 95

D

data
    printing the value of 87
    visualizing 85
Data Browser 41
    related options 86
    using 85
databases, setting option for preprocessors 103
debugging
    fully linked executables 90
    multiple processes 91
    techniques 84 to 91
    threaded processes 91
demand-driven code generation 63
    and header files 50
dependencies, in CL targets 66, 68
designing, CL targets 66
destructors, examining 89
display, setting your iv
dual language support 95

E

editing, search path used when 101
editor, specifying iv
eight_bit option 103
enhancing your application 93
environment variables
    DISPLAY iv
    EDITOR iv
environment, setting up 99 to 107
Error Browser 79
    using to suppress warnings 81
errors
    large numbers of 81
    responding to 80
    run-time 78
    saving to file 74
examining
    classes 86
    variables with the Workspace 85
executables, debugging 90
executing
    main() 77
    your program with different test data 96
execution
    continuing 80
    interrupting 90
    tracing 89
expand command 89
expressions, displaying 85
EZSTART, loading your application with 70

F

file suffixes
    searched for 103
files
    .ocenterinit 104
    ocenter.proj 71
    ocenterinit 104
    .octrusrcmd 105
    swapping 82
fork system call 91
fully linked executables, debugging 90

functions
  CenterLine 88
  creating 106
  implicit 88
  printing the number of calls to 87
  resolving undefined 95

G

-**G** switch, for loading files 49
-**g** switch, overriding 49
global variables, initializing 96

H

header files
  loading 50
  resolving problems locating 74
  search path for 48
history, **history** command 94

I

-**I** switch, for loading files 48
implicit function calls 88
incremental linking 92
info command 89
Inheritance Browser
  components of 86
  printing contents 86
instrumented object files
  run-time error checking 78
  speed of execution 79
  swapping 82
interpreter, using 93

L

-**L** switch, for loading files 48
-**l** switch
  and **load_flags** option 63
  for loading libraries 49
languages, dual support for 95
leaks, finding memory 77
libraries
  attaching 49
  linking from 51
  resolving problems in loading 74
  search path for 48
link command 73
link -list command 73
linking
  and template instantiation 51
  from libraries 51
  incrementally 92
listing, search path used when 101
load command 73
load_flags option 48
load_header command 50
loading
  additional documentation on 51
  C files 50, 82
  header files 50
  object files that don't exist 49
  object files using CL targets 68
  object files with debugging information 49
  object files without debugging information 49
  project files 72
  search path used when 101
  setting load switches with options 64
  source files 50
  switches 48
  the Bounce program 47 to 51
  with CL targets 66 to 69
  with EZSTART 70
  with the **load** command 65
  your application 61 to 74

M

m4 preprocessor, setting options for 103
macros, in CL targets 69
main() function, executing 77
make command 69, 70
makefiles
  and EZSTART 70
  and loading your application 63
  and reloading files 92
  maintaining 72
memory leaks, finding 77
memory locations, monitoring 90
messages, saving to file 74
metacharacter expansion, preventing 68
multiple processes, debugging 91

N

name, of program, setting 77

O

object files
  reloading 92
  run-time error checking 79
  swapping 82
  that don't exist 49
  and watchpoints 90
  with debugging information 49, 65, 69
  without debugging information 49, 65, 69
ObjectCenter
  loading your application 61 to 74
  running your application 77 to 98
  setting up your environment 99 to 107
ocenter.proj file 71
.ocenterinit file 104
ocenterinit file 104
.octrusrcmd file 105
options
  ansi 103
  batch_load 91
  cxx_suffixes 103
  eight_bit 103
  for setting load switches 64
  load_flags 48
  path 101
  preprocessor 103
  print_inherited 86
  print_pointer 86
  print_runtime_type 86
  print_static 86
  program_name 103
  save_memory 81
  saving 71, 104
  setting 99 to 107
  setting in CL targets 68
  show_inheritance 86
  swap_uses_path 83
  that affect Data Browser 86
  unset_value 81
Options Browser 101

P

parent processes, setting breakpoints 91
path used for loading, listing, editing, swapping 101
pdm
  additional documentation on 91
  setting executable name 103
performance issues 63
Postscript printing 86
precompiled header file 63
preprocessors, setting option for 103
print_inherited option 86
print_pointer option 86
print_runtime_type option 86
print_static option 86

printing
number of calls to a function 87
the value of data structures 87
printing from Browsers 86
program_name option 77, 103
Project Browser
exporting contents to a file 98
setting options 101
project file
definition 71
for saving options 104
loading 72
saving 71
projects
and CL targets 72
saving 71
Properties windows 101
prototyping, in the Workspace 93

R

rcs
creating functions for 106
setting resources for 106
rebuilding
object files 49
your project 92
regular object files
run-time error checking 78
swapping 82
reloading files 92
revision control systems
creating functions for 106
setting X resources for 106
run command 77
Run Window 77
running
test suites 96
your application 75 to 98
run-time error checking
discussion 78
reducing 81
run-time problems, responding to 80

S

save command 71
save_memory option 81
saving
error messages to file 74
option settings 104
projects 71
your option settings 104
sccs
creating functions for 106
setting resources for 106
setopt command 64
sh command 68
shell commands
creating 105
in CL targets 66, 68
show_inheritance option 86
signals, saving 71
Source area
listing the corresponding source file 79
source command 94
source files
loading 50
reloading 92
run-time error checking 78
swapping 82
standard targets, building 69
start command 96
startup files
.ocenterinit 104
for saving options 104
stop on command 90
suffixes searched for 103
suppressing, warnings 80
suppressions, saving 71
swap command 80, 82, 83
swap_uses_path option 83

swapping
- files 82
- search path used when 101

switches, for loading 48
system testing 96

T

tabs, in CL targets 68
targets
- CL, *See* CL targets
- standard 69

template code 33
terminal emulators, and custom commands 105
test suites, running 96
testing
- automating 91
- running test suites 96
- system testing 96
- unit testing 95
- your application 93

threaded processes, debugging 91
tracing execution 89
troubleshooting
- **load** and **link** commands 73
- run-time issues 97
- the **swap** command 83

U

undefined symbols, listing 73
unit testing 95
unload command 92
unres command 73
unset_value option 81

V

variables, examining with the Workspace 85

W

warnings
- ignoring 80
- large numbers of 81
- responding to 80
- run-time 78
- saving suppressions 71
- saving to file 74
- suppressing 80

watchpoints 90
win_io option 77
Workspace
- additional documentation on 85
- commands, *See* commands
- directing output to 77
- entering code fragments 85
- prototyping in 93
- setting options in 101 to 102
- unit testing in 95
- using 84

X

X resources, setting 106
xterm client, and Run Window 77