
CodeCenter Tutorial

Version 4.1.1

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 CodeCenter solely to facilitate the use of CodeCenter 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

CodeCenter is an interactive 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 CodeCenter 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 CodeCenter 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 CodeCenter environment. It also shows you how to load your own code into CodeCenter, 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 69 after reading this introduction.

What you should know before starting

This guide doesn't assume any knowledge of CodeCenter, 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 CodeCenter

Before you start using CodeCenter, choose the editor CodeCenter 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 107 describes some other options and customizations you may want to set up when you are more familiar with CodeCenter.

Specifying your editor

CodeCenter 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 CodeCenter. See 'Using your own editor' on page 115 for more information.

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

Setting your display

Before you invoke CodeCenter, 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 CodeCenter

You invoke CodeCenter with the **codecenter** command.

```
% codecenter
```

By default, CodeCenter 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 CodeCenter, you see the Main Window, which acts as the hub of the CodeCenter 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. CodeCenter'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 CodeCenter windows from the Browsers menu.

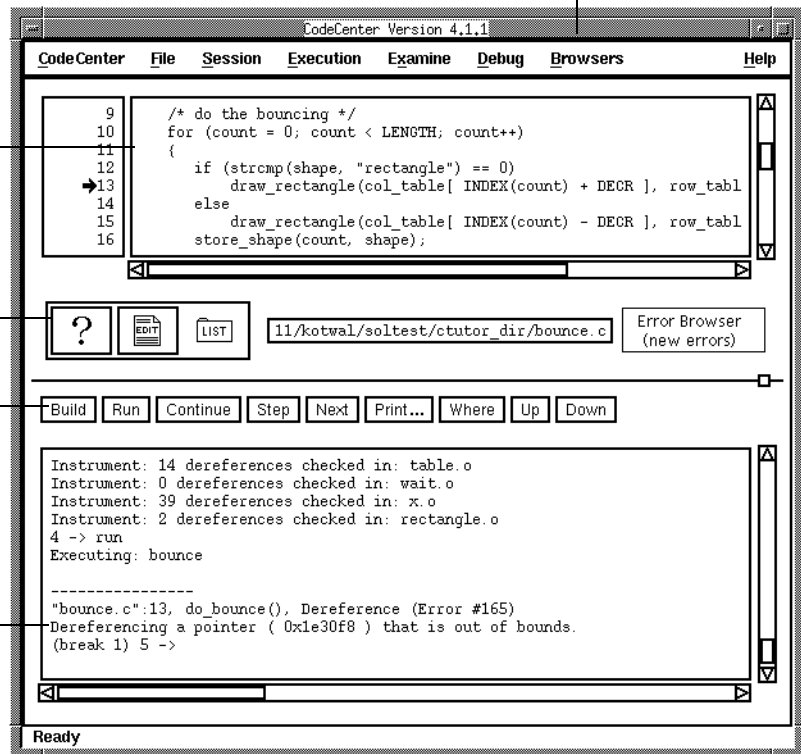
Browsers menu

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 CodeCenter 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 *CodeCenter User Guide* and *CodeCenter 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 *CodeCenter Reference* by typing **man** in the CodeCenter 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 CodeCenter 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: <pre style="margin-left: 40px;">-> help email</pre>
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 <pre style="margin-left: 40px;">-> sh man cldoc</pre> <p>You may need to set your MANPATH environment variable before starting CodeCenter. For example:</p> <pre style="margin-left: 40px;">% setenv MANPATH path/CenterLine/man:\$MANPATH</pre>

How to use the tutorial

To explore the CodeCenter 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 tutorial, you will fix some problems, then revise the program so it bounces a circle.

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

- **Debugging a corefile.** You use the CodeCenter process debugging mode with a fully linked executable.
- **Correcting run-time and static errors.** You set up a CodeCenter project and use component debugging mode to track down a run-time error and make the correction.
- **Enhancing a program interactively.** You use code visualization to analyze the structure of the program, dynamic debugging items to explore the flow of control in the code, and the interpreter for prototyping.
- **Loading the Bounce program with the load command.** You practice loading a program into CodeCenter without a makefile.

The first three tasks are designed to be performed in sequence; however, you can omit the first task if you prefer. The last task can be completed independently.

Setting up the tutorial directory

To set up the tutorial directory, go to your home directory and invoke the `ctutor` command, as follows:

```
% cd
% ctutor
```

If the operating system does not find the `ctutor` 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 `ctutor` command creates a directory called `ctutor_dir` in the current directory and copies the tutorial files to the new directory. The `ctutor_dir` contains the following files:

Makefile	circle.c	shape.c.dump	wait.c
bounce.c	main.c	shape.c.good	x.c
bounce.c.bad	rectangle.c	shape.h	x.h
bounce.h	shape.c	table.c	

Contents

Using this book iii

Part I Tutorial 1

Chapter 1 Debugging a corefile 3

Getting started 5

Specifying a corefile as a target for CodeCenter 6

Finding and fixing the error that causes the segmentation fault 7

Examining data structures using the Data Browser 9

Building and running the new executable 11

Chapter 2 Correcting run-time and static errors 13

Starting Chapter 2 15

Setting up a CodeCenter project for Bounce 15

Viewing the project components 16

Finding the problem using run-time error checking 18

Swapping a file from object to source form 22

Exploring the code to understand the problem 23

Fixing the error 25

Using load-time error checking to clean up the code 27

Using the Error Browser 28

Saving a project file 30

Chapter 3 Enhancing a program interactively 31

Starting up and loading your project 33

Adding a new module to your project 34

Examining program elements with the Contents window 35

Looking at the program's structure 36

Listing a function in the Source area 40

Using interactive debugging items to follow execution 41

Calling a function from the Workspace 45

Examining a linked list dynamically 47

Modifying the program 53

Chapter 4 Using the load command 57

Starting Chapter 4 59

Setting options 61

Loading the Bounce program 62

Linking from libraries 64

Part II Getting started with your own code 65

Chapter 5 Loading your application 67

- Getting ready to load your code 69
- Loading your application with the load command 71
- Loading your application with CenterLine makefile targets 73
- Creating a makefile with the EZSTART utility 77
- Saving your project 78
- Troubleshooting the load and link commands 80

Chapter 6 Running your application 83

- Running your program 85
- What is run-time error checking? 86
- Adding more run-time error checking to object files 87
- Responding to run-time problems 88
- Swapping a file from object to source 90
- Debugging techniques 92
- Rebuilding your project 99
- Exploring, enhancing, and testing your application 100
- Troubleshooting run-time issues 104

Chapter 7 Setting up your environment 107

- Setting options in the Workspace 109
- Saving your option settings 112
- Customizing your environment 113

Index 117

List of Tables

- Table 1 Resolving **load** and **link** Problems 80
- Table 2 Types of Run-Time Error Checking 86
- Table 3 Troubleshooting Swapping 91
- Table 4 Troubleshooting Run-Time Issues 104
- Table 5 Additional Workspace Options 111

List of Tips

- Saving load-time error messages to a file 81
- Exporting the contents of the Project Browser to a text file 105

Part I: Tutorial

This part of the manual contains a tutorial that takes you on a tour of the CodeCenter environment. The tutorial has four 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:

- *Debugging a corefile*
- *Correcting run-time and static errors*
- *Enhancing a program interactively*

Work through the last chapter to learn how to use the load command to get code into CodeCenter.

Chapter 1 Debugging a corefile

For the purposes of this tutorial, the Bounce program represents code that you have inherited. In exploring this program, you will see that the code has a range of problems, from glaring to inconspicuous. We can use CodeCenter to correct these problems.

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:

```
% cd ~/ctutor_dir
```

Compiling the Bounce program at the shell

The first step in exploring the Bounce program is simply to run it. This involves building the executable by using the UNIX **make** utility at the shell, as follows:

```
% make bounce_dump
```

If the name of the makefile target arouses your suspicions about the current state of the Bounce program, you are on the right track.

Running an executable that dumps core

As it stands, the Bounce program will generate a core dump. To run the Bounce program and dump a core file, use the following command at the shell:

```
% bounce_dump
```

The Bounce program opens an execution window, draws the first rectangle, and then crashes. You should receive a system error notifying you of a segmentation fault or bus error and a core dump. The size of the corefile for **bounce_dump** varies from platform to platform. On the Sun platform, you may need to issue the following command before running **bounce_dump**:

```
% unlimited coredumpsize
```

Starting CodeCenter in process debugging mode

To debug a core file, CodeCenter must be in process debugging mode. Enter the **codecenter** command with the **-pdm** switch to start CodeCenter in process debugging mode:

```
% codecenter -pdm
```

The Workspace displays a startup message for process debugging mode, and the Workspace prompt (pdm 1 ->) indicates that CodeCenter is in process debugging mode.

Specifying a corefile as a target for CodeCenter

You use CodeCenter's process debugging mode to work with a fully linked executable such as **bounce_dump**. Process debugging mode allows you to target a fully linked executable in several ways: the executable alone, the executable together with a corefile, or the executable as a running process. Since you have a corefile for **bounce_dump**, you can target the executable together with the corefile, which is an image of the process memory for **bounce_dump** at the point where the fault occurred.

To do this, type the following command:

```
pdm 1 -> debug bounce_dump core
Debugging program `bounce_dump'
Core was generated by `bounce_dump'.
Program terminated with signal 11, Segmentation
fault.
#0  0x23a4 in store_shape (count=0, shape=0xf7fff4f0
"rectangle") at shape.c:11
    11          *old = *new;
```

NOTE The output of commands may vary from platform to platform. The examples and 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*.

In response to the **debug** command, the Workspace displays messages about the executable and corefile and identifies the line of code that generated the segmentation fault or bus error. The Workspace also shows that line 11 generates the error with the assignment ***old = *new**.

Finding and fixing the error that causes the segmentation fault

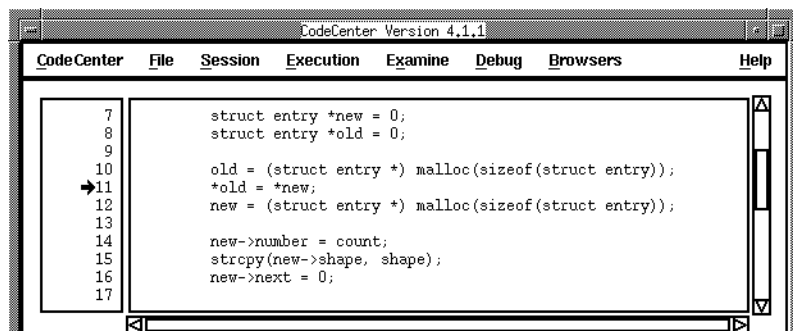
Process debugging mode is similar to a standard symbolic debugger like **gdb** in following execution, setting breakpoints, stepping through code, and printing the values for expressions. CodeCenter places these same facilities within its integrated programming environment and graphical user interface. For example, in addition to simply printing values, you can use the Data Browser to examine graphical representations of your data structures. This section shows how you can exploit CodeCenter's capabilities in process debugging mode.

Stopping execution at the point of the segmentation fault

To stop execution at the point of the segmentation fault and see this error in context, you simply select the **Run** button above the Workspace.

The Bounce window appears with a single rectangle in it. Then, CodeCenter automatically stops execution at the point of the error, shows the corresponding source code in the Source area, and indicates where execution stopped with the Execution symbol. CodeCenter also generates a break level in the Workspace.

Examining the lines of source code surrounding the error, you can recognize that the assignment of ***new** to ***old** is misplaced. This line should follow the allocation of memory and assignment of values for **new**.



The screenshot shows the CodeCenter interface with a menu bar (CodeCenter, File, Session, Execution, Examine, Debug, Browsers, Help) and a source code editor. The code is as follows:

```
7      struct entry *new = 0;
8      struct entry *old = 0;
9
10     old = (struct entry *) malloc(sizeof(struct entry));
11     *old = *new;
12     new = (struct entry *) malloc(sizeof(struct entry));
13
14     new->number = count;
15     strcpy(new->shape, shape);
16     new->next = 0;
17
```

Line 11 is highlighted with a right-pointing arrow, indicating the location of the segmentation fault.

Editing the source code

Now that you see that the problem is the misplaced assignment `*old = *new`, you can fix it by invoking your editor on that specific line, as follows:

- 1 In the Source area, move the mouse pointer over the number 11 at the left margin.
- 2 Press the Right mouse button and select **Edit line 11** from the pop-up menu.

This opens your editor on the source file, with the cursor placed at the beginning of the line containing the error. Fix the error by doing this:

- 1 In your editor, move line 11 below line 16 (`new->next = 0;`). Your code should now look like the following:

```
old = (struct entry *) malloc(sizeof(struct entry));
new = (struct entry *) malloc(sizeof(struct entry));

new->number = count;
strcpy(new->shape, shape);
new->next = 0;
*old = *new;
```

- 2 Save this file.

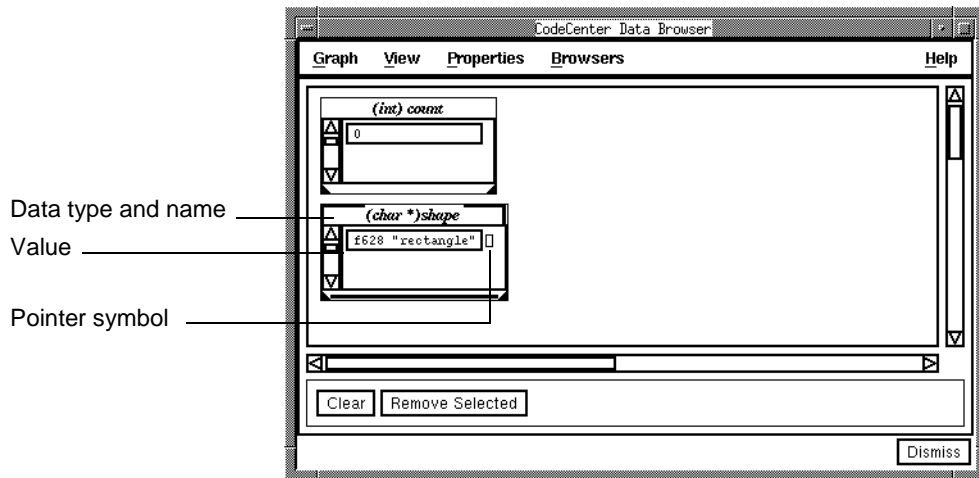
Examining data structures using the Data Browser

Now that you have located and fixed the immediate problem, you can use the opportunity to examine various data structures. For example, you can examine the formal argument **count** in the following way:

- 1 In the Source area, use the mouse to highlight the string **count**.
- 2 In the Main Window, display the **Examine** menu and select **Display**.

You can repeat these same steps for the string **shape**.

The Data Browser opens and displays a graphical representation for the variables **count** and **shape**:



Using spot help to understand the displayed item

To understand the displayed items shown in the Data Browser, use CodeCenter's spot help:

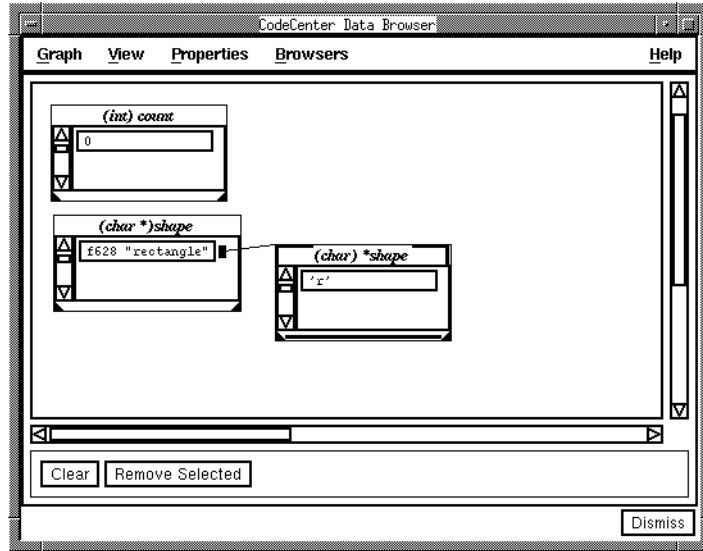
Move the mouse pointer over the name **shape** in the Display area and press the **F1** key.

Dereferencing a displayed pointer

Since **shape** is a pointer to **char**, the graphical item shows a Pointer symbol at the right of the value. You can use this box to display the dereferenced value:

At the right side of the **shape** display item, select the Pointer symbol.

The Pointer symbol is filled in; a reference line is drawn from the box to a graphical representation for the character pointed to by **shape**:



To exit the Data Browser, select **Dismiss** at the bottom of the window.

Building and running the new executable

To try out the changes you have just made to the code for **shape.c**, you first need to rebuild the executable. We have provided a new makefile target that will pick up your changes, give the executable a more appropriate name, and remove the previous executable and corefile. Before you continue, reset the Workspace to the top level:

```
pdm (break 1) 2 -> reset
```

Using the make command in the Workspace

When CodeCenter is in process debugging mode, the **make** command invokes the UNIX **make** utility. Rebuild the executable and give it a new name by using the following command:

```
pdm 3-> make bounce_nodump
```

Running the fixed executable

You now have a new executable named **bounce_nodump**. Use the **debug** Workspace command as follows to make this executable the new debugging target. Then use the **run** command to execute the program.

```
pdm 4-> debug bounce_nodump  
debug: Deleting all debugging items.  
Debugging program 'bounce_nodump' (previous program  
'/net/paris/kotwal/ctutor_dir/bounce_dump')  
pdm 5-> run  
Executing: /net/paris/kotwal/ctutor_dir/bounce_nodump  
Program exited normally.  
Resetting to top level.
```

The Bounce program now runs to completion; your correction has removed the condition that caused a core dump. However, Bounce does not behave as gracefully as you may have hoped; the rectangle makes erratic jumps, rather than bouncing in smooth curves. This indicates that there is some other problem involved. This problem occurs during run time but does not generate a segmentation fault or bus error.

You can continue with the next chapter and find and fix the problem, but if you want to stop at this point, enter the **quit** command in the Workspace:

```
pdm 6 -> quit
```


Chapter 2 Correcting run-time and static errors

Now that you have the Bounce program executing to completion, you need to turn your attention to correcting its behavior. For some reason, the bounce becomes erratic near the end of the program.

In this chapter, you will use CodeCenter's component debugging mode to track down a run-time error, understand the code, and make the correction. Component debugging mode allows you to work with a project made up of any combination of source, object, and library file components, rather than a fully linked executable.

Starting Chapter 2

If you didn't try the first chapter, or if you quit CodeCenter after completing it, start CodeCenter in component debugging mode:

```
% codecenter
```

If you are continuing from the first chapter, switch CodeCenter from process debugging mode to component debugging mode:

- 1 In the Main Window, display the **CodeCenter** menu and select **Restart Session**.
- 2 Check that the Component Debugger radio button is selected.
- 3 At the bottom of the dialog box, select **Restart Debugger**.

In the Main Window, the Workspace prompt no longer includes **pdm**, indicating that CodeCenter is now in component debugging mode. When you switch debugging modes, CodeCenter clears the Source area.

Setting up a CodeCenter project for Bounce

To work in CodeCenter's component debugging mode, you begin by establishing a project. To do this, you load in the source, object, and library files related to the application you are working on.

CodeCenter automatically loads the standard C library (**libc**). On some platforms, CodeCenter loads the shared version of this library.

Using a makefile to set up a project

You have several options for loading the files you need for your project. For this tutorial, we have prepared a makefile target that will take care of loading the proper files to set up your project. To invoke it, type the following Workspace command:

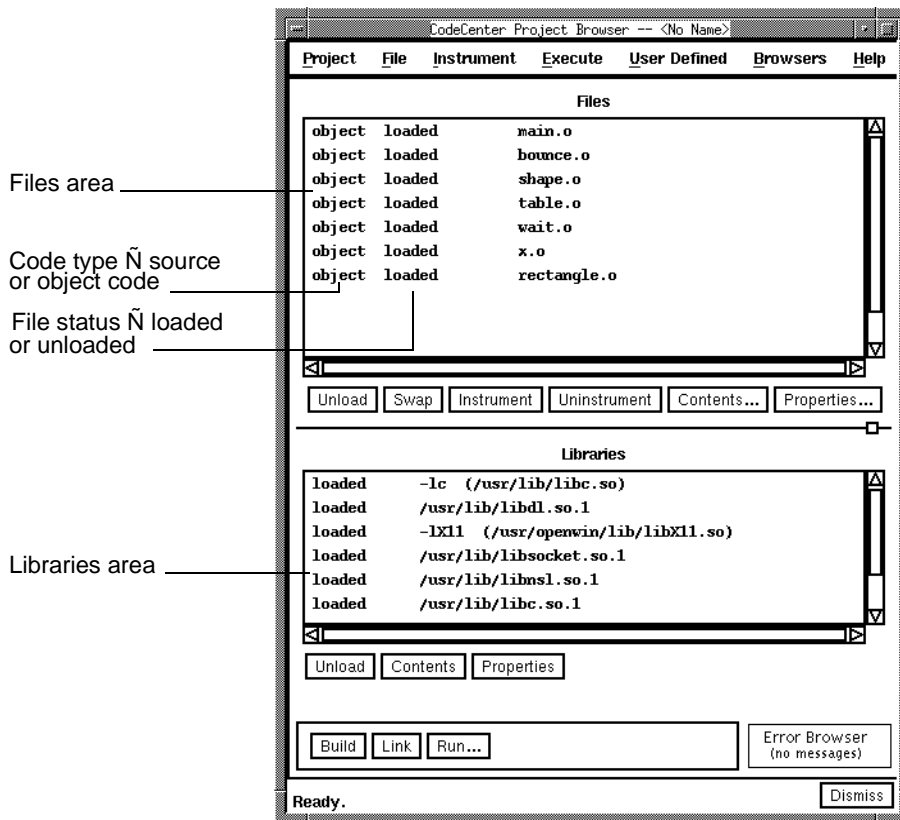
```
1 -> make bounce_project
```

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

Viewing the project components

CodeCenter lists all the components of the current project in the Project Browser. To view the project components, display the **Browsers** menu and select **Project Browser**.

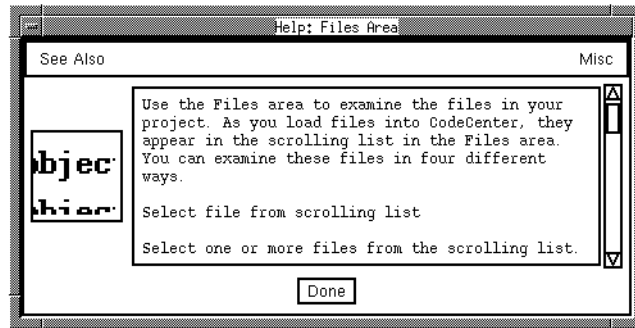
The Project Browser shows the components of your project in two categories: files (either source or object code) and libraries.



The makefile target loaded all the Bounce program files in object form. For the fastest load time, you typically load most files in object form; you load a file in source form only while you are actively developing that module. Once a file is loaded, you can swap between object and source forms at any time.

Using spot help to understand the Project Browser

To understand how to work with files shown in the Files area of the Project Browser, you can use CodeCenter's spot help feature. Move the mouse pointer into the Files area and press the **F1** key.



Running the program in component debugging mode

Now that you have established the project for the Bounce program, you can run it by doing the following:

- 1 At the bottom of the Project Browser, select the **Run** button.
- 2 At the bottom of the dialog box, select the **Run** button.

To put the Project Browser aside while you turn your focus back to the Main Window, you can move the Browser to the side or iconify it.

CodeCenter supports two different types of object files: regular and instrumented. *Instrumented object files* have run-time error checking enabled in them, which means that error-checking code is inserted in the object files in memory. Because all the files of this project are currently loaded in regular object form, the program runs in the same way and at the same speed as the fully linked executable in process debugging mode.

In the next section, we'll use instrumented object files.

Finding the problem using run-time error checking

To track down the problems with the program's behavior, you need to enable CodeCenter's run-time error checking. While instrumented object code does not offer the full range of error checking that CodeCenter provides for source code, it does provide most of the run-time error checking and gives execution speed closer to regular object code.

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 *Reference*, or check your *Release Bulletin*. If instrumentation isn't supported on your platform, go on to 'Swapping a file from object to source form' on page 22.

Instrumenting the object files

The first step is to enable run-time checking for the object files in the project by instrumenting these files with additional debugging information. To instrument all the object files in the current project, use the following Workspace command:

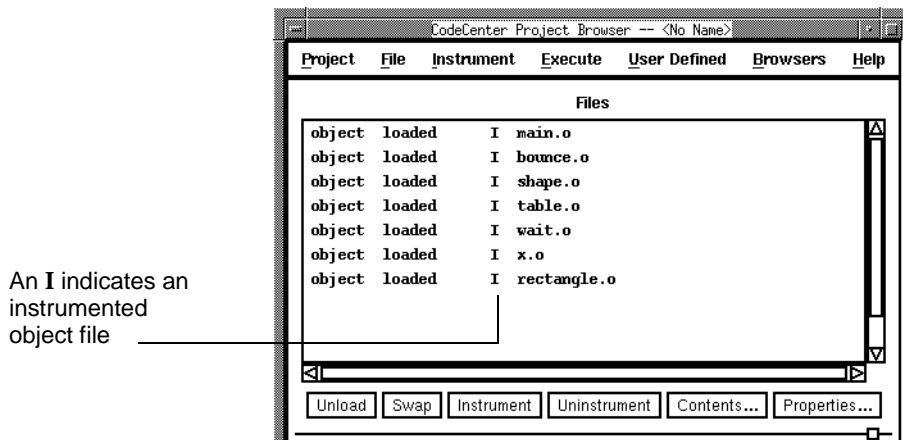
```
2 -> instrument all
Loading:
/u6/CenterLine/c_4.0.0/<arch>/misc/ocode/instrument.o
Instrument: 7 dereferences checked in: main.o
Instrument: 4 dereferences checked in: bounce.o
Instrument: 11 dereferences checked in: shape.o
Instrument: 14 dereferences checked in: table.o
Instrument: 0 dereferences checked in: wait.o
Instrument: 39 dereferences checked in: x.o
Instrument: 2 dereferences checked in: rectangle.o
```

The Workspace shows each file as it is instrumented and reports the number of dereferences checked in for that file.

NOTE

For the purposes of this tutorial, we use the **instrument all** command. However, this might not be the best approach to use when instrumenting your own code. For example, if you have many object files loaded, you may want to instrument a subset of the files. For more information, see the **instrument** entry in the *CodeCenter Reference*.

The Project Browser shows an **I** to the left of each filename to indicate that these files are now instrumented:



Running the Bounce program with instrumented object code

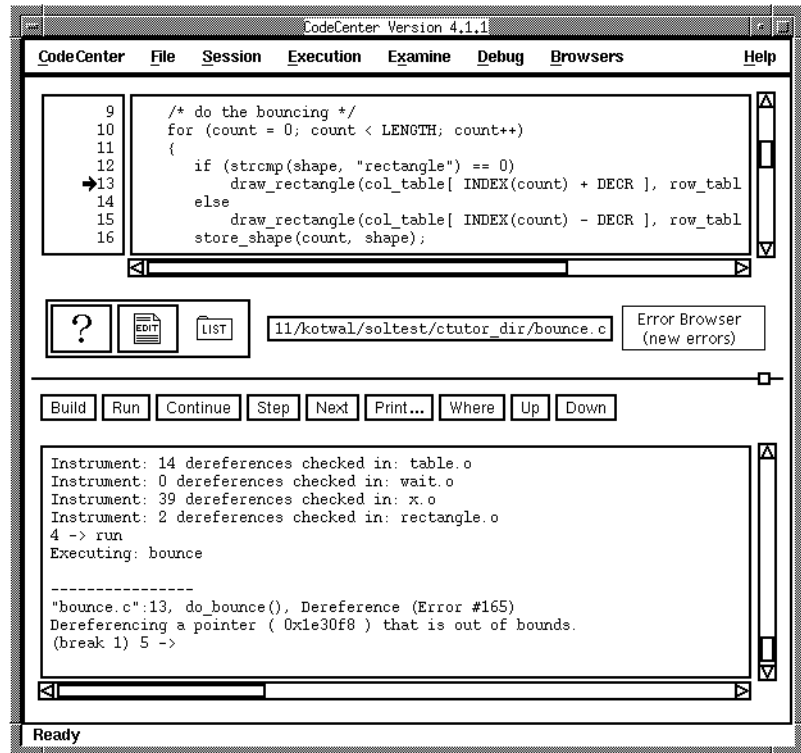
When you run a program in component debugging mode, CodeCenter automatically checks for run-time errors in source files and instrumented object files.

To run the program, select the **Run** button in the middle of the Main Window.

The Bounce program starts running. Because CodeCenter is now checking for run-time violations in the instrumented object code, the execution is slightly slower than with regular object code.

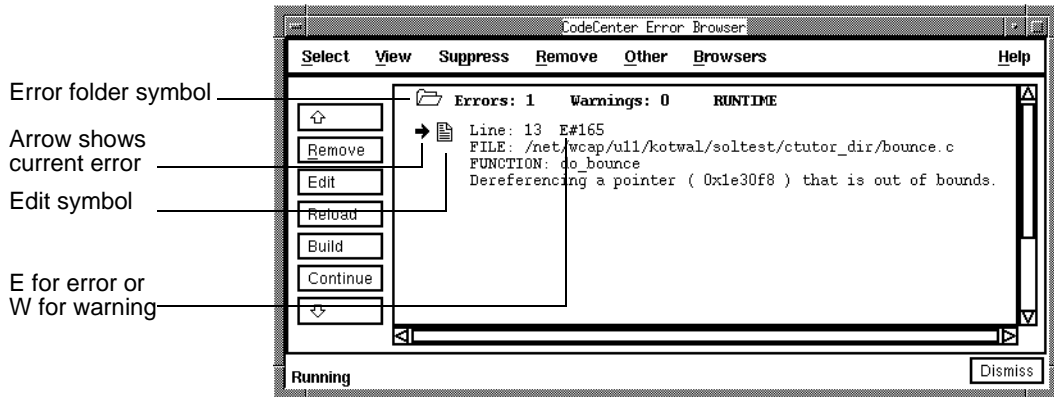
After bouncing the rectangle for a while, CodeCenter stops the execution of the Bounce program at a different place because it detects an error. CodeCenter lists the source file causing the error (**bounce.c**) in the Source area and indicates the line where execution stopped (line 13) with the execution symbol.

CodeCenter 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.



Viewing error messages in the Error Browser

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 CodeCenter automatic run-time error checking in component debugging mode .

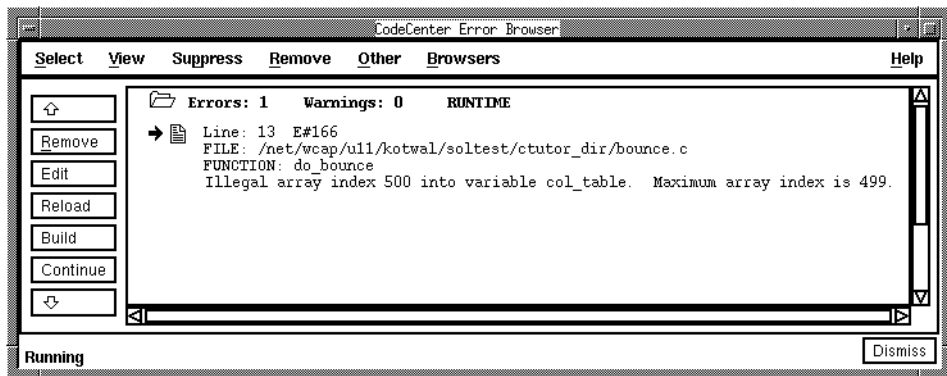
Swapping a file from object to source form

Using CodeCenter's run-time error checking with instrumented object code, you have located an out-of-bounds error on line 13 of the source file for **bounce.o**. However, to take full advantage of the information available through CodeCenter's interpreter and debugger, you can swap this file from object to source form. Since you have narrowed your focus down to a single module, the greater error checking for source code is well worth the cost of slower execution speed in this file. To swap **bounce.o** to source form, display the **File** menu in the main Window and select **Swap**.

CodeCenter unloads **bounce.o** and loads **bounce.c**. Now with the problematic file loaded in source form, you can execute the program to get more help in tracking down the problem.

In the middle of the Main Window, select the **Run** button.

This time, CodeCenter stops execution at line 13 in **bounce.c** and, as shown below, the Error Browser shows a more informative error message that there is an illegal index of 500 into the array **col_table**. The message further notes that the maximum index is 499.



Exploring the code to understand the problem

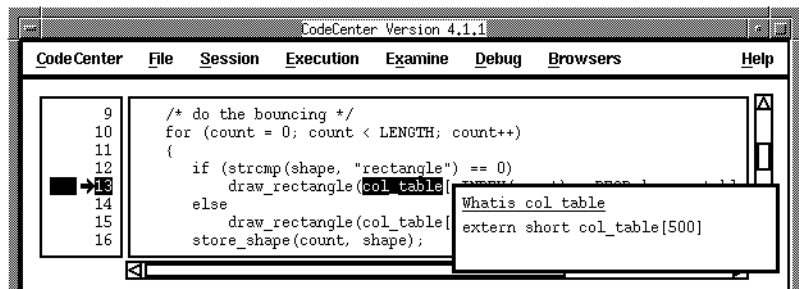
You now have the run-time error located and have the problematic file loaded in source form. Now you are ready to investigate the relevant code carefully so you can understand how to fix the problem.

Viewing the definition of an identifier

CodeCenter provides a quick, point-and-click way to get information about the definition of an identifier in your code. CodeCenter then displays information about the expression in a pop-up window. To view the definition of the identifier `col_table` at line 13, in the Source area do the following:

- 1 Select the string `col_table` by highlighting it or by simply moving the mouse pointer over it.
- 2 Press and hold the Shift key and click the Middle mouse button.

The Whatis window opens and displays the definition of `col_table` as an array of **extern short** with 500 elements:



To dismiss the Whatis window, click the Left mouse button.

You can also get this same type of information using the **whatis** command in the Workspace:

```
(break 1) 5 -> whatis DECR
#define DECR (100)
```

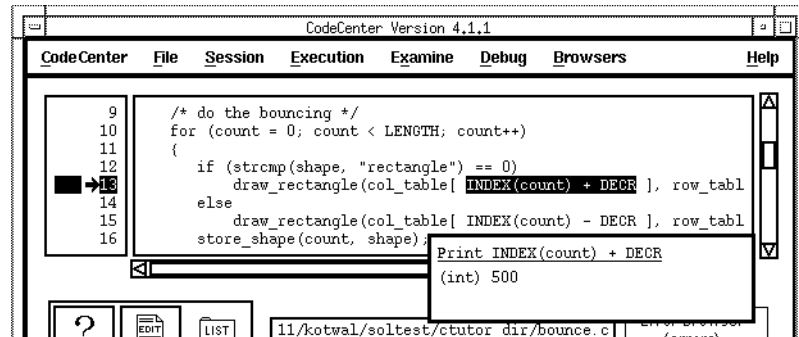
The **whatis** command shows that **DECR** is a **#define** macro with the value 100.

Viewing the value of an expression

You can use a similar point-and-click method to discover the run-time value of a variable or an expression. To do this:

- 1 Select the string **INDEX(count) + DECR** by dragging the mouse pointer.
- 2 Press the Shift key and click the Left mouse button.

The Print window opens and shows that the current value of the expression **INDEX(count) + DECR** is 500:



To dismiss the Print window, click the Left mouse button.

In the same way, you can print the value of each of the components of this expression. The value of `count` is 300 and the value of **INDEX(count)** is 400.

Based on the values that you have discovered, it becomes clear that the index expression **INDEX(count) + DECR** is coming up with an illegal index for `col_table` because the decrement is being *added*, rather than subtracted. Comparing this index expression with a similar one for `row_table` on the same line highlights the problem.

Fixing the error

Now that you have found the error in **bounce.c**, you can use CodeCenter's integration with your editor to streamline the process of correcting the code. To invoke your editor on the source code containing the error, do the following:

- 1 At the left margin of the Source area, move the mouse pointer over number 13 and press the Right mouse button.
- 2 Select **Edit line 13**.

This opens your editor on the source file, with the cursor automatically placed at the beginning of the line containing the error. Now do the following to fix the error:

- 1 In your editor, change line 13 so that **DECR** is subtracted, not added. The line will then look like the following:

```
draw_rectangle(col_table[ INDEX(count) - DECR ],
```

- 2 In your editor, save the file.

Rebuilding and running your corrected program

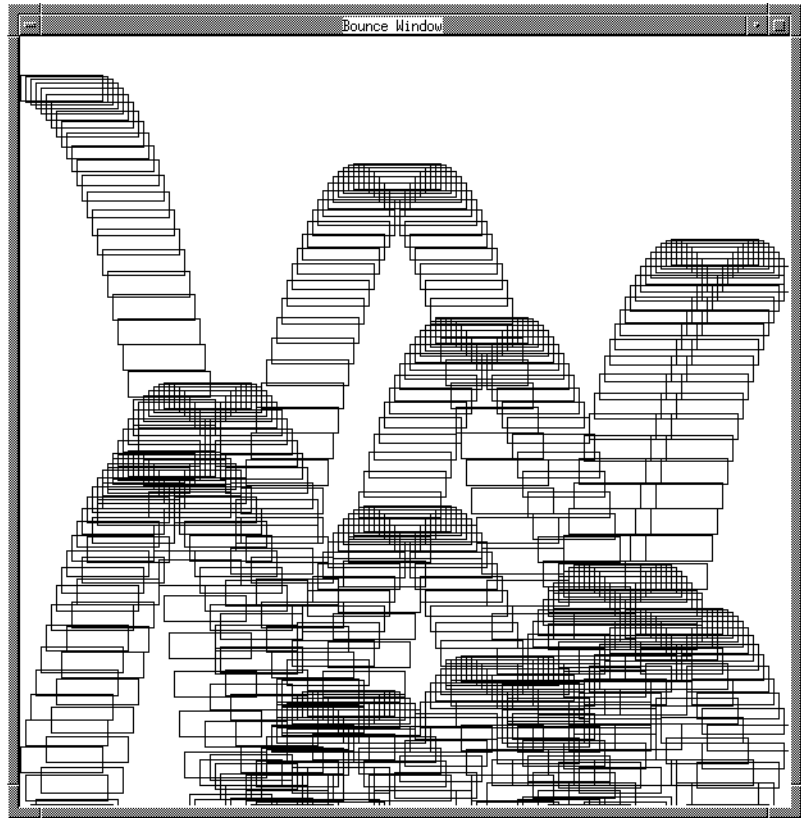
You have changed a source file for a component of your project. You now need to bring your project up to date by rebuilding the project:

In the middle of the Main Window, select the **Build** button.

CodeCenter incrementally links and reloads only the file that you changed rather than all the files. In addition, CodeCenter automatically tracks all dependencies. Your correction is now integrated into your project and the modified source code appears in the Source area. To see if this correction fixes the bounce behavior, do the following:

In the middle of the Main Window, select the **Run** button.

The Bounce program runs to completion with no run-time violations, and the rectangle bounces in a smooth curve all the way, as shown:



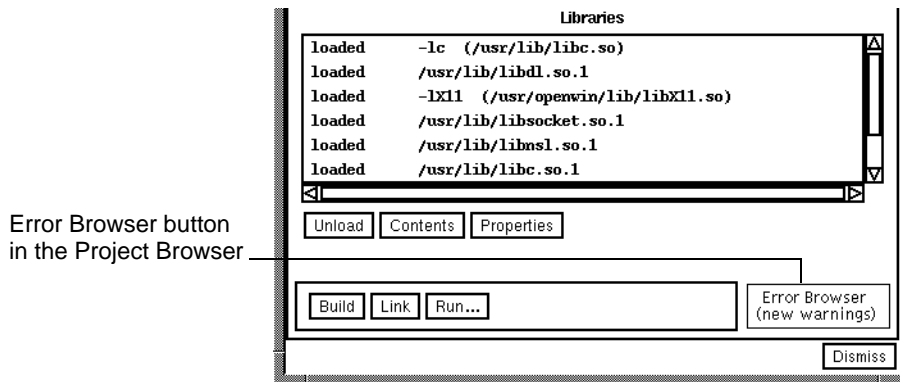
Using load-time error checking to clean up the code

You now have the Bounce program running correctly. However, having functionally correct code is not necessarily the same thing as having clean code. CodeCenter's load-time error checking provides a powerful tool for automating the code cleanup phase in your programming cycle.

Any time you load source code, CodeCenter does extensive lint-style checking for correct syntax and code usage. For example, to see if **shape.c** is clean, you would simply swap **shape.o** from object to source form by doing the following:

- 1 In the Project Browser Files area, select the line containing **shape.o**.
- 2 In the middle of the Project Browser, select the **Swap** button.

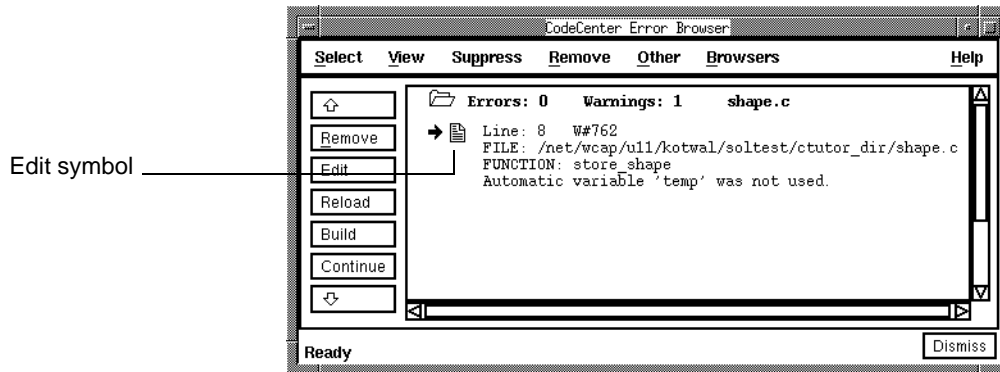
As CodeCenter loads **shape.c**, it finds load-time violations and signals the presence of load-time warnings with the **Error Browser** button, as shown:



Double-click the **Error Browser** button. This opens the Error Browser, with a Messages line for **shape.c** that shows no errors and one warning.

Using the Error Browser

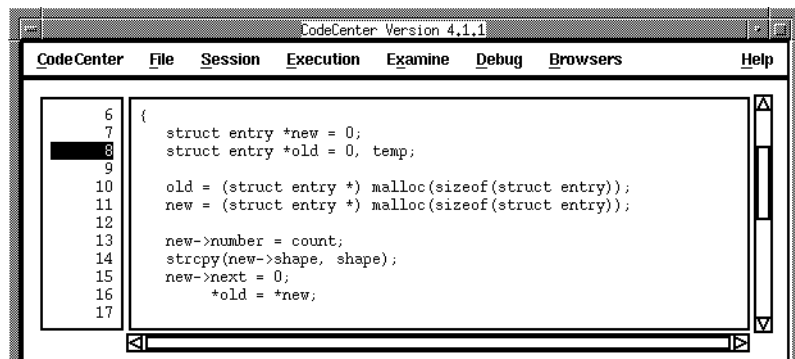
Click on the error folder symbol to see the warning message: the automatic variable **temp** is not used in the function.



The Error Browser allows you to work with load-time and run-time violations in several ways, such as listing and calling your editor on the source code corresponding to a violation message.

When you open a file of violations in the Error Browser, the first violation message is automatically selected. Since **shape.c** only has one message, the warning message you are interested in is already selected.

To view the line of source code that corresponds to the warning message, look at the line number that is highlighted in the Source area.



Viewing the offending line in context makes it clear that the variable **temp** is simply a relic of earlier coding and should now be eliminated. To invoke your editor on the source file containing the violation, select the Edit symbol in the Error Browser.

Your editor may ask you if you want to load the **shape.c** file from disk since it was updated when you built the **bounce_project** target at the beginning of this chapter. Respond yes if prompted by your editor.

Editing your code

Your editor is automatically positioned at the source line containing the load-time problem. To fix the problem:

- 1 Delete the string **temp** from line 8 so the code looks like the following:

```
{
    struct entry *new = 0;
    struct entry *old = 0;
```

- 2 In your editor, save the file.

Reloading the corrected code

To update your CodeCenter project, you need to reload **shape.c**, which you have just modified. To reload **shape.c**, click the Reload button in the Error Browser. No new error messages appear in the Error Browser, indicating that the source code of **shape.c** now passes CodeCenter's load-time checking and ensuring that the code is clean.

In some situations, you may want to suppress the reporting of certain classes of warnings and save suppressions across sessions or projects. You'll learn about this later, in 'Responding to run-time problems' on page 88.

Saving a project file

In the next chapter, you will again be working closely with **bounce.c** and **shape.c**; therefore, you can simply leave them in source form when you save your project. To easily pick up your work in the next chapter where you left off, you can save your current setup in a project file by doing the following:

- 1 In the Main Window, display the **CodeCenter** menu and select **Save Project**.
- 2 In the Save Project dialog box, edit the **Save to File** line to the name **bounce.proj**.
- 3 At the bottom of the dialog box, select the **Save Project** button.

You can also save your project from the **Project** menu in the Project Browser, or with the **save** command in the Workspace.

Quitting CodeCenter

You have now completed the second chapter. Because the next chapter involves loading your project file at the start of a session, you need to leave CodeCenter at this point. To do so:

- 1 In the Main Window, display the CodeCenter menu and select **Quit** CodeCenter.
- 2 In the dialog box, select **Quit**.

As you see in the Quit Verification dialog box, you can save a project file when you leave CodeCenter. Since you have just taken care of this in the Project Browser, there is no need to save your project again when you exit this time.

NOTE The next chapter uses instrumentation, which 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 *Reference*, or check your *Release Bulletin*.

If instrumentation isn't supported on your platform, you have now completed the tutorial. You may want to go on to 'Loading your application' on page 67 to learn how to load your own code into CodeCenter.

Chapter 3 Enhancing a program interactively

Now that you have the Bounce program executing correctly, you can turn your attention to adding an enhancement. In this chapter, you will use CodeCenter's component debugging mode to explore the code further and enhance the program to bounce either a rectangle or a circle, based on a command-line argument.

Starting up and loading your project

You need to start CodeCenter in component debugging mode and re-establish the Bounce project that you set up in the second chapter. To restart, enter the **codecenter** command at the shell with the name of your project as an argument:

```
% codecenter bounce.proj
```

Re-establishing your project inside CodeCenter

Starting up with your project name as an argument to the **codecenter** command re-establishes your project automatically. If you prefer, you can re-establish the Bounce project after starting CodeCenter:

1 Enter the **codecenter** command at the shell without an argument:

```
% codecenter
```

2 In the Main Window, display the **Browsers** menu and select **Project Browser**.

3 In the Project Browser, display the **Project** menu and select **Load Project**.

4 At the **Load From File** input line, enter the name **bounce.proj**.

5 At the bottom of the dialog box, select **Load Project**.

Adding a new module to your project

There is a file in the tutorial directory that provides code for drawing a circle, rather than a rectangle. The first step in using this code to enhance the Bounce program is to add the file **circle.o** to the current project. To do this:

- 1 In the Project Browser, display the **Project** menu and select **Add Files**.
- 2 In the Files list of the dialog box, move the mouse pointer over the file **circle.o** and double click the Left mouse button. This adds **circle.o** to your project.
- 3 To dismiss the dialog box, select **Done**.

You have added **circle.o** to your project, and you can now instrument it to enable run-time error checking for this file by doing the following:

- 1 In the Files area of the Project Browser, select **circle.o**.
- 2 Below the Files area, select the **Instrument** button.

Understanding the program's structure

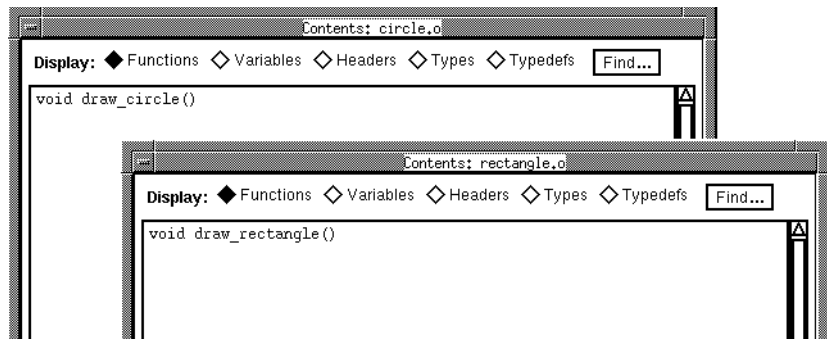
Before changing the existing program to incorporate the new module to draw a circle, you might want to take advantage of CodeCenter's tools for understanding the underlying structure of the program that makes up your current project.

Examining program elements with the Contents window

To understand program elements at the file level, CodeCenter has a Contents window that gives you various views on the data in a given file. You can choose to examine any of the following elements: functions, variables, headers, types, or typedefs. For example, you can use the Contents window to examine the contents of **circle.o**, which you just added to your project, and compare it with **rectangle.o**:

- 1 In the Files area of the Project Browser, select the filenames **circle.o** and **rectangle.o**. You can select more than one filename by holding down the Control key while you click on the second filename.
- 2 At the bottom of the Files area, select **Contents**.
- 3 Move the two Contents windows so you can see the contents of both windows (they may open so one is hiding the other). You might also want to resize the windows to get them side by side.

You can see that **rectangle.o** has one function **draw_rectangle()**, and **circle.o** has one function **draw_circle()**.



To see the other program elements for each file, you simply change the display selection above the Contents area. To view the variables in **circle.o** and **rectangle.o**, select **Variables** in each Contents window. In the same way, compare the contents for **Headers**, **Types**, and **Typedefs**. When you have finished comparing different elements for these two files, select the **Dismiss** button at the bottom of each window.

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

To get a different structural view of the Bounce program, you can use the Cross-Reference Browser to view the calling hierarchy and references to global symbols in the program. In the Project Browser, display the **Browsers** menu and select **Cross-Reference Browser**.

The Cross-Reference Browser opens. You can specify any function as the focus point for cross-referencing. For example, if you cross-reference **draw_rectangle()**, the Browser will show references from functions and global variables that reference **draw_rectangle()** and all functions and global variables that are called by it. To cross-reference **draw_rectangle()** and specify it as the focus function for cross-referencing, do the following:

NOTE Before opening a dialog box, CodeCenter will attempt to use the current X11™ selection. If there is a current selection in *any* X window, you may get an error message instead of the dialog box.

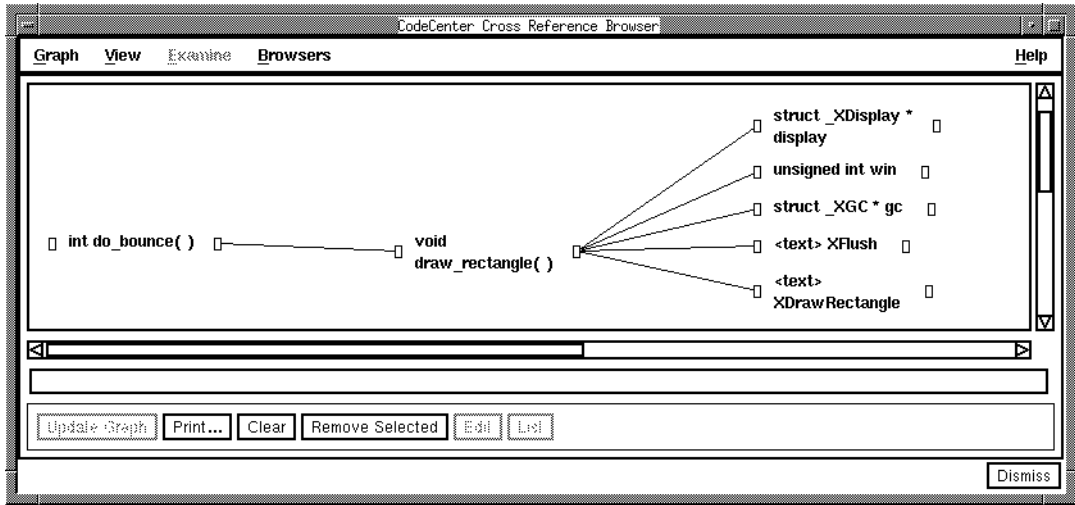
If this happens, clear the selection by clicking on white space in an X terminal window.

- 1 In the Cross-Reference Browser, display the **Graph** menu and select **Cross Reference Symbol**.
- 2 At the **XRef Symbol** input line, type **draw_rectangle**.
- 3 Select the **Cross Reference** button.

As seen in the next illustration, the Reference area of the Cross-Reference Browser shows **draw_rectangle()** as a function referenced by **do_bounce()** and referencing two library functions (**XFlush()** and **XDrawRectangle()**) and three global variables (**display**, **win**, and **gc**).

If type information is available for a function, the Cross-Reference Browser shows the function's return type; for example, **void draw_rectangle()**. If an object file does not contain type information for an item, instead of showing a return type, CodeCenter lists the area

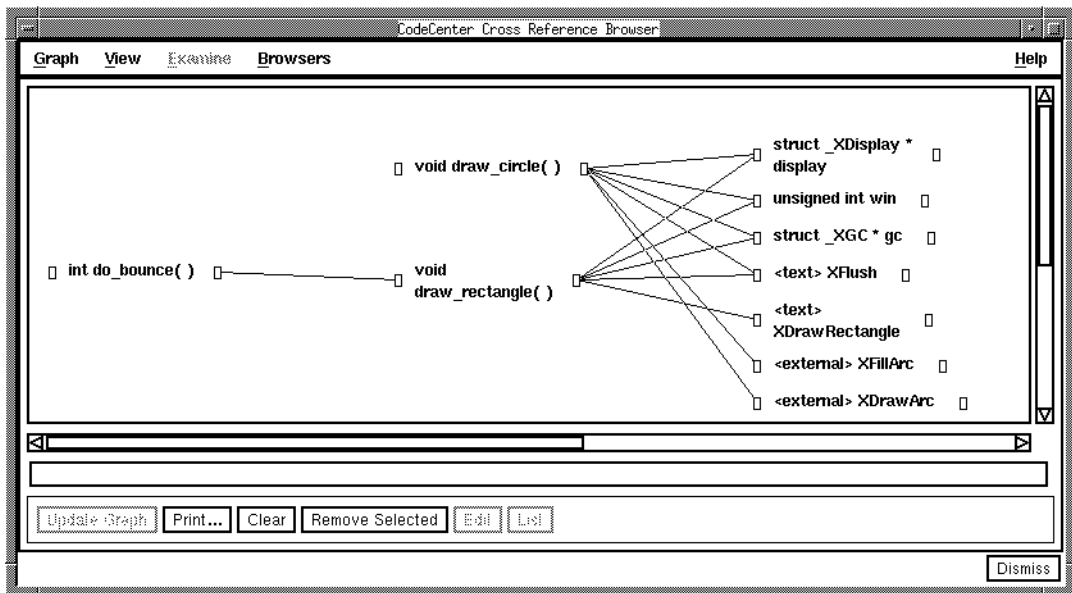
in the object file in which the item is located: `<text>` or `<data>`; for example `<text> XFlush()`. If the object file does not contain enough information even to determine the location of the item, then the type is listed as `<extern>`.



To compare the reference structure of `draw_rectangle()` with the structure for `draw_circle()`, do the following:

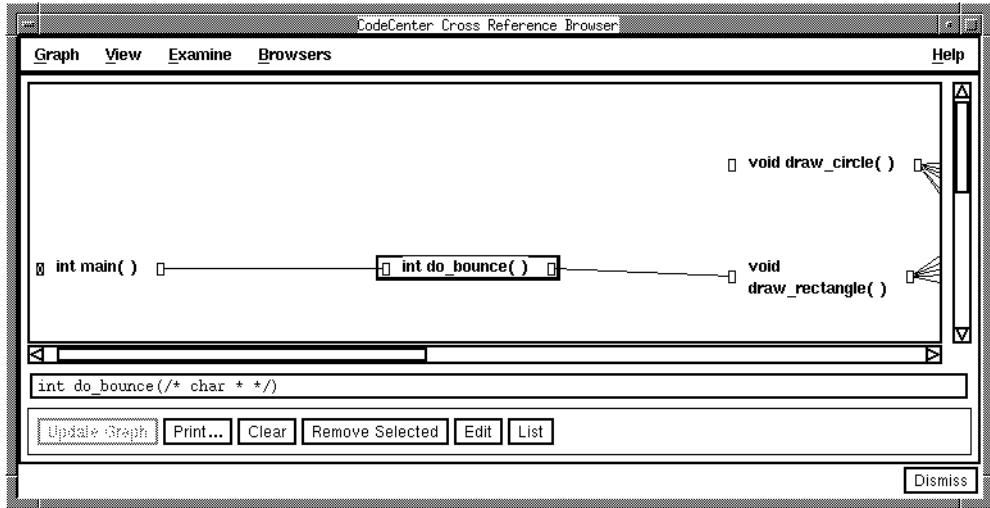
- 1 In the Cross-Reference Browser, display the **Graph** menu and select **Cross Reference Symbol**.
- 2 In the input line of the dialog box, type the string `draw_circle`.
- 3 At the bottom of the dialog box, select the **Cross Reference** button.

The Reference area shows that `draw_circle()` references most of the same items as `draw_rectangle()`, but it is not referenced anywhere in the program.

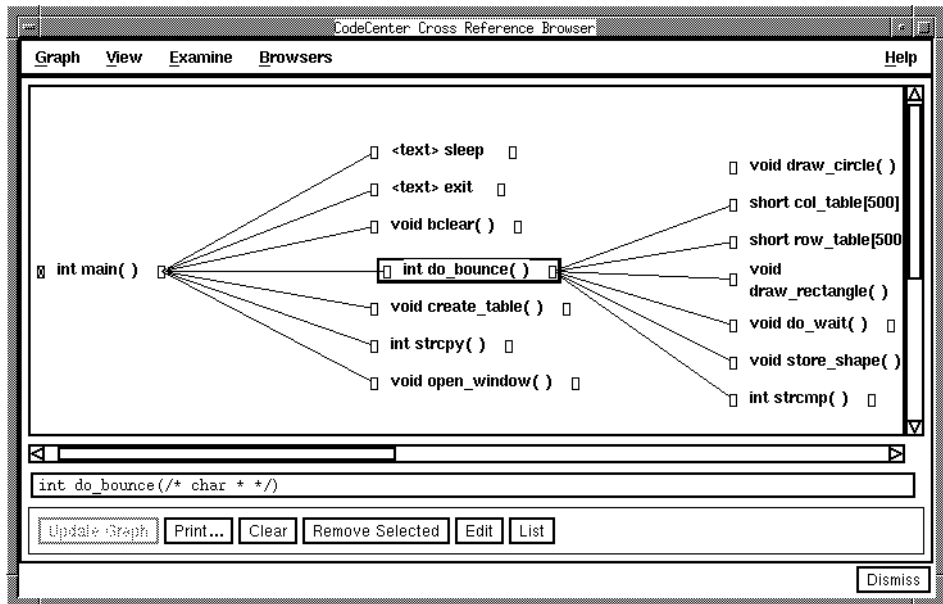


The Cross-Reference Browser allows you to trace the calling structure by selecting the boxes on the left and right sides of reference items. To follow the call structure back from `do_bounce()`, select the box to the left of `do_bounce()`.

The `do_bounce()` function becomes the new focus function and shows a reference line coming to it from `main()`. Use the scrollbars to adjust the view in the Reference area.



To understand the overall calling structure of the program, you can select the boxes to the right of `main()` and `do_bounce()`. The box to the right indicates what functions or global variables the function references.



Listing a function in the Source area

Since you have found out that `draw_circle()` is not being called and that `draw_rectangle()` is a similar function that is called by `do_bounce()`, you might suspect that `do_bounce()` is also where you will want to integrate `draw_circle()` into the Bounce program. To understand the code in `do_bounce()`, you can first examine it by listing it in the Source area, as follows:

- 1 In the Reference area of the Cross-Reference Browser, move the mouse pointer over the `do_bounce()` reference item. The mouse pointer must be over the white space *between* the function name and the Pointer symbol on the left or right of it.
- 2 Press the Right mouse button and select **List**.

The Source area of the Main Window lists the source code for `do_bounce()`. You can also list `do_bounce()` by typing the following in the Workspace.

```
-> list do_bounce
```

The source code reveals that there are two places where `draw_rectangle()` can be called. Which call is made depends on the value of `shape`, which is passed in as an argument when `do_bounce()` is called from `main()`. In `main()`, `shape` is set by a command-line argument or else defaults to the string `rectangle`.

Using interactive debugging items to follow execution

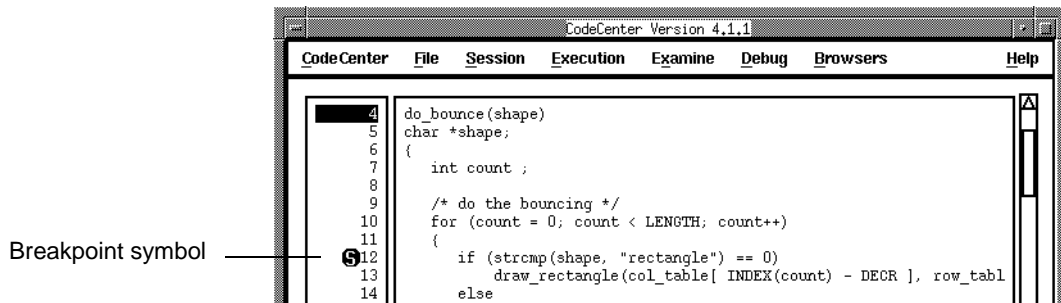
Now that you have a sense of the calling structure, you can use CodeCenter's debugging features to follow the flow of control interactively as the program executes.

Setting a breakpoint

CodeCenter allows you to specify a breakpoint where you want execution to stop. To set a breakpoint at the `if` statement that calls `draw_rectangle()`, do the following:

- 1 Move the mouse pointer over the line number **12** at the left of the Source area and press the Right mouse button. The pop-up menu lists actions on **line 12**.
- 2 On the pop-up menu, select **Set Breakpoint Here**.

A Breakpoint symbol appears to the left of the line number:



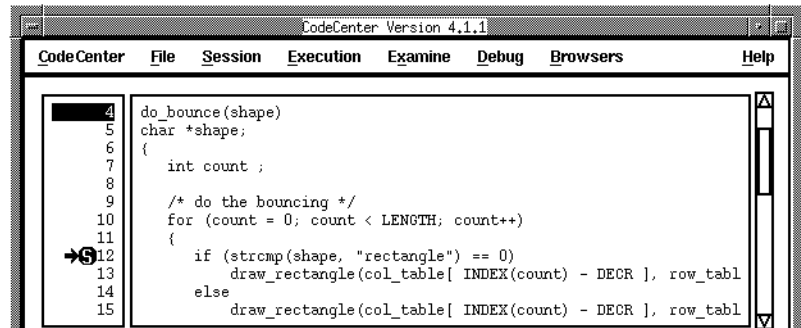
As a shortcut, you can set a breakpoint by selecting the line number in the Source area. You can delete a breakpoint by selecting the Breakpoint symbol.

Executing to a breakpoint

Now when you execute the Bounce program, CodeCenter will stop execution at line 12 in **bounce.c** and establish a break level in the Workspace.

In the middle of the Main Window, select the **Run** button.

In the Source area, the Execution symbol indicates that execution has stopped at the breakpoint on line 12:



Continuing execution from a breakpoint

By examining the source code at this breakpoint, you can see that this is the statement that draws each rectangle that forms the bounce. You might, therefore, want to simply move the execution through to this point several times to get a sense of where each of these rectangles will be drawn.

To continue execution back to this breakpoint, select **Continue** in the Button Panel in the main Window.

The first rectangle appears in the execution window, and execution is again stopped at line 12.

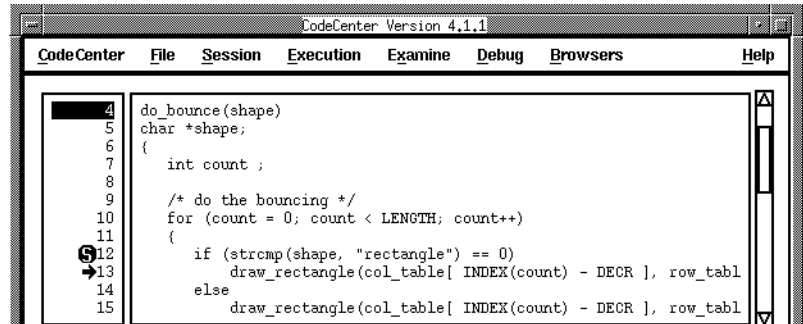
To have the Bounce program draw a second rectangle, select the **Continue** button again.

Single stepping execution

To follow execution more closely, you can single step through each statement.

- 1 In the middle of the Main Window, select the **Step** button.

The statement at line 12 is executed, and the Execution symbol moves to line 13.



```

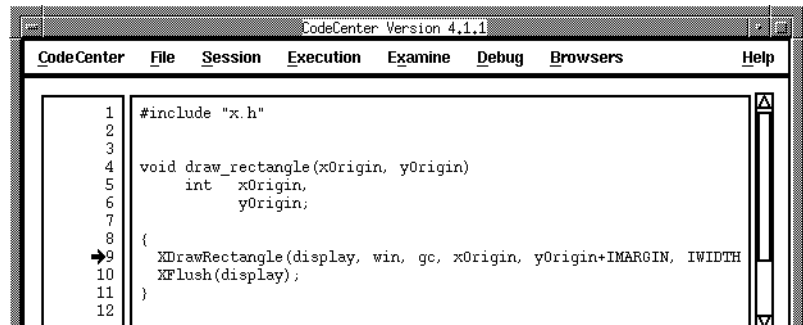
CodeCenter Version 4.1.1
CodeCenter  File  Session  Execution  Examine  Debug  Browsers  Help

4 do_bounce(shape)
5 char *shape;
6 {
7     int count ;
8
9     /* do the bouncing */
10    for (count = 0; count < LENGTH; count++)
11    {
12        if (strcmp(shape, "rectangle") == 0)
13            draw_rectangle(col_table[ INDEX(count) - DECR ], row_tabl
14        else
15            draw_rectangle(col_table[ INDEX(count) - DECR ], row_tabl

```

Execution is now at line 13, which is a call to **draw_rectangle()**. If you single step on this statement, you will follow execution into the function that is called.

- 2 In the middle of the Main Window, select the **Step** button again.



```

CodeCenter Version 4.1.1
CodeCenter  File  Session  Execution  Examine  Debug  Browsers  Help

1 #include "x.h"
2
3
4 void draw_rectangle(xOrigin, yOrigin)
5     int xOrigin,
6         yOrigin;
7
8
9     XDrawRectangle(display, win, gc, xOrigin, yOrigin+IMARGIN, IWIDTH
10    XFlush(display);
11
12

```

- 3 To single step through `draw_rectangle()`, in the middle of the Main Window, select the **Step** button three times.

Execution returns from `draw_rectangle()`, and the Execution symbol is now at line 16 in `do_bounce()`, as shown:

The screenshot shows the CodeCenter interface with the following code and execution state:

```

11 {
12     if (strcmp(shape, "rectangle") == 0)
13         draw_rectangle(col_table[ INDEX(count) - DECR ], row_tabl
14     else
15         draw_rectangle(col_table[ INDEX(count) - DECR ], row_tabl
16     store_shape(count, shape);
17     do_wait();
18 }
19 return 0;
20 }
21
22

```

The execution symbol (a circle with a vertical line) is positioned to the left of line 12. A right-pointing arrow is positioned to the left of line 16, indicating the current execution point.

The next statement is a call to the function `store_shape()`. Rather than single stepping through this function, you can execute it and move directly to the next statement in the current function.

- 4 In the middle of the Main Window, select the **Next** button.

In one move, execution has returned from `store_shape()` and the Execution symbol is now at line 17.

The screenshot shows the CodeCenter interface with the following code and execution state:

```

11 {
12     if (strcmp(shape, "rectangle") == 0)
13         draw_rectangle(col_table[ INDEX(count) - DECR ], row_tabl
14     else
15         draw_rectangle(col_table[ INDEX(count) - DECR ], row_tabl
16     store_shape(count, shape);
17     do_wait();
18 }
19 return 0;
20 }
21
22

```

The execution symbol (a circle with a vertical line) is now positioned to the left of line 17, indicating that execution has moved to the next statement in the function.

Calling a function from the Workspace

In addition to stepping through the execution path, while at a break level you can also follow a new execution path dynamically by calling a function from the Workspace. For example, the next statement in the execution path is line 17 of **bounce.c**, a call to **do_wait()**. But, as an experiment, you might want to explore **store_shape()** instead. First, to set a breakpoint in **store_shape()** so you can step through the new execution path, do the following:

- 1 In the Main Window, display the **Debug** menu and select **Set Breakpoint**.
- 2 At the **Function** input line, type the string **store_shape**.
- 3 Select the **Set Breakpoint** button.

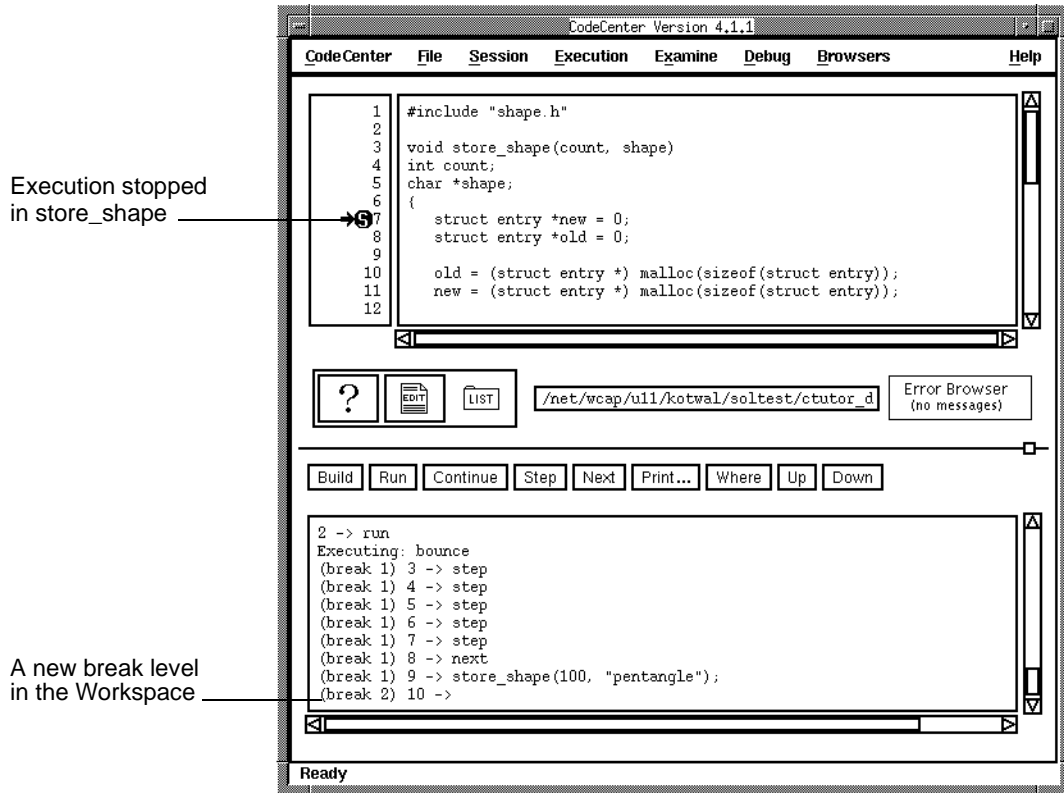
The breakpoint is set in **shape.c**, which is not listed in the Source area.

Second, to call **store_shape()** from the Workspace and experiment with the actual arguments, you enter the following C statement (including the semicolon at the end of the line):

- 1 In the Workspace, type in the following C statement:

```
(break 1) 9 -> store_shape(100, "pentangle");
```
- 2 Press the Return key to enter this statement at the Workspace.

The Source area shows that execution is now stopped at line 7 in **shape.c**, which is the first statement in **store_shape()**:



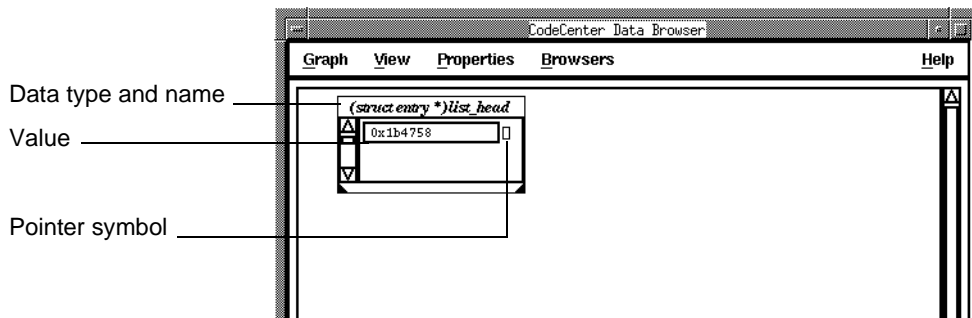
The Workspace prompt shows that execution is stopped at break level 2. This is because the breakpoint at line 7 of **shape.c** is in a different execution path from the one for break level 1 where you made the call to **store_shape()**.

Examining a linked list dynamically

Now that you are stopped at the first line of `store_shape()`, you can use the opportunity to examine various data structures dynamically. The most significant data structure in `store_shape()` is the linked list pointed to by `list_head`. To display the head of the linked list (`list_head`), do the following:

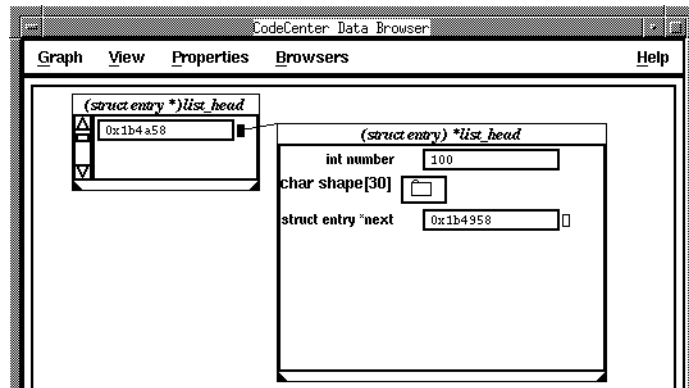
- 1 In the Source area, select the string `list_head` on line 18 by dragging the mouse pointer from one end of the string to the other.
- 2 Display the **Examine** menu and select **Display**.

The Data Browser opens and displays a graphical representation for `list_head`, as shown:



The variable `list_head` is a pointer, and the display item for it shows a Pointer symbol. You can use the Pointer symbol to see the first node in the list. In the Data Browser, select the Pointer symbol at the right side of the display item's value field.

The dereference box is darkened and a reference line is drawn from the box to a graphical representation for the **struct** pointed to by **list_head**.

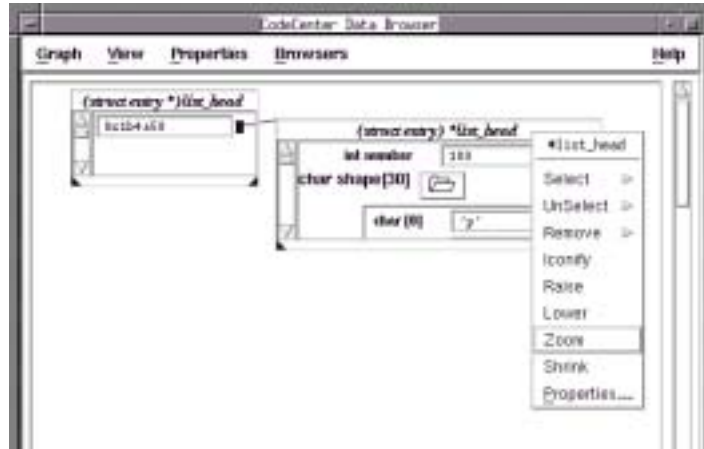


The **struct** pointed to by **list_head** has three elements: **number**, **shape**, and **next**. The Folder symbol to the right of the **char** array **shape** indicates that there are additional subelements that you can view.

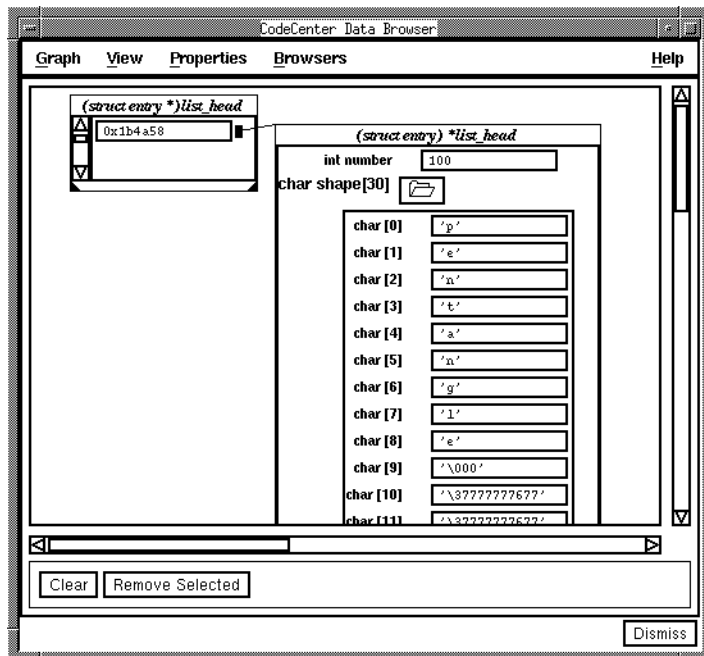
To view the subelements of **shape**, select the Folder symbol to the right of **shape**.

The Folder symbol opens and the array elements are displayed. Since most of the elements are out of sight, you need to zoom the size of the graphical item for ***list_head** and scroll the zoomed item into view with the vertical scrollbar. To zoom the size of the graphical item, do the following:

- 1 In the Data area, move the mouse pointer over the graphical item for ***list_head** and press the Right mouse button. The mouse pointer must be immediately to the left or right of the name **(struct entry) *list_head**.
- 2 In the pop-up menu, as shown, select **Zoom** with the Right mouse button.



When you select **Zoom**, the following happens.

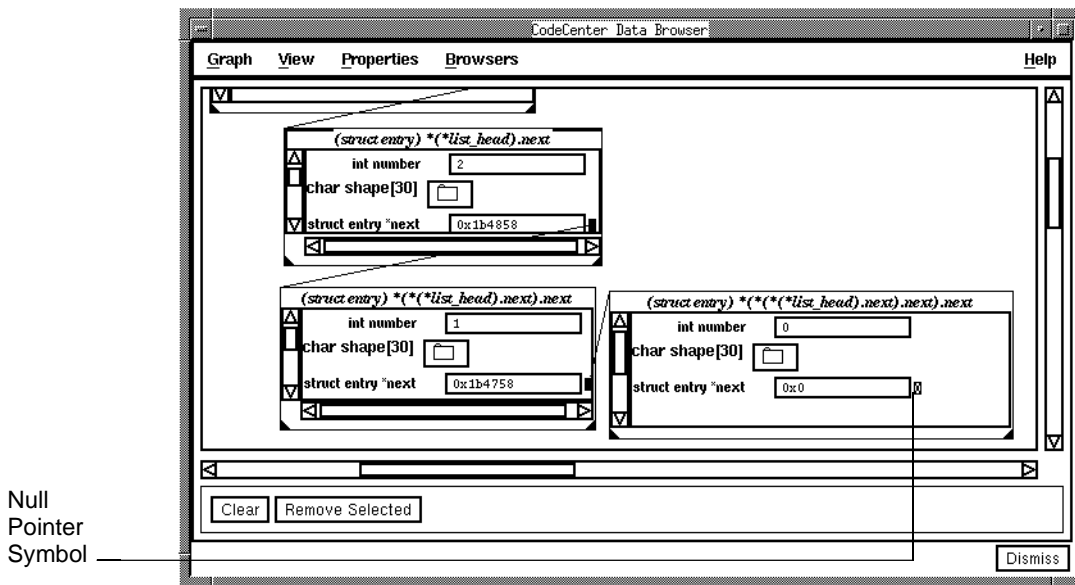


You can now see that the **shape** array contains the string **rectangle**. To save display space in the Data Browser, you can hide the subelements once you have viewed them. To hide the subelements in the **struct** ***list_head**, select the open Folder symbol to the right of **shape**. This closes the folder and removes the array elements from the display. To shrink the data item, select **Shrink** from the pop-up menu, or use the resize corners on the data item.

Following the linked list

So far you have displayed the pointer to the list and first node of the list. To follow the linked list, you simply select the reference box for the first node in the same way you did with **list_head**. To display the next node in the linked list, do the following:

- 1 In the Data Browser, select the Pointer symbol in the first list node.
- 2 Select the Pointer symbol in the next node.
- 3 Use the horizontal and vertical scrollbars to bring in view the display items you are interested in.



The Pointer symbol for the third node in the list is filled in with the Null Pointer symbol, which indicates a null pointer that cannot be dereferenced.

Seeing how the program uses old and new

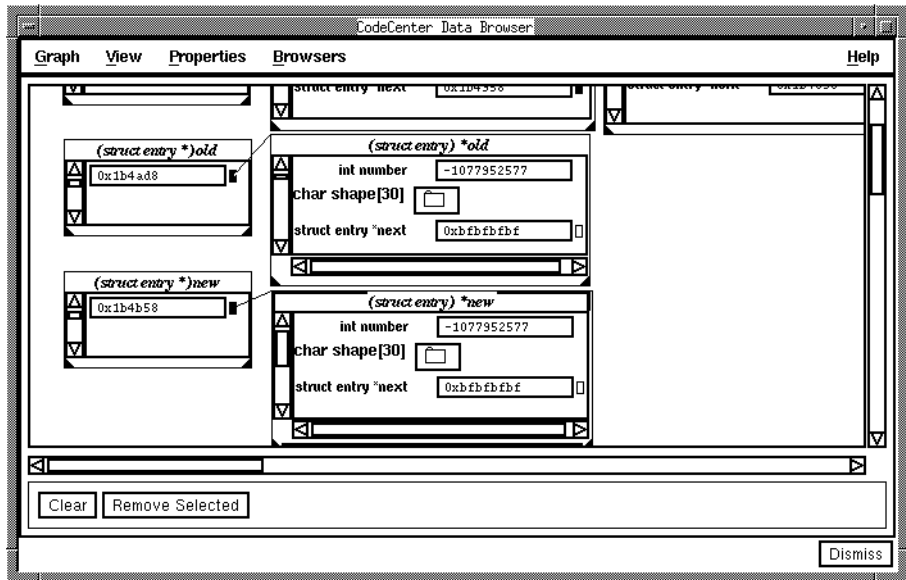
Do the following to begin stepping through **store_shape** line by line to see the flow of execution:

- 1 In the Main Window, select the **Step** button.
- 2 Repeat stepping to line 13, beyond the allocation of memory for **old** and **new**.

You can use the Data Browser to display a graphic representation of how the program uses **old** and **new** in adding a new node to the list. To display the pointers **old** and **new**, do the following:

- 1 Make sure you have no string selected in the Source area of the Main Window.
- 2 In the Data Browser, display the **Graph** menu in the menu bar and select **New Expression**.
- 3 At the input line of the dialog box, type the string **old**.
- 4 At the bottom of the dialog box, select the **Data Browse** button.
- 5 Repeat steps 2 to 4 for the string **new**.
- 6 To dereference each pointer, move the mouse pointer over the Pointer symbol and click the Left mouse button.

In addition to the linked list of nodes pointed to by **list_head**, the Data Browser now shows **old** and **new** and the newly allocated display items they point to. Besides scrolling or resizing the window to get the best view, you can also move the displayed items by selecting and dragging them with the mouse to new locations.



Watching a new node get added

By stepping through the rest of the function, you can watch the new node being added. To do this:

- 1 In the middle of the Main Window, select the **Step** button.
- 2 Repeat this until execution moves to line 26.

As you step through the function, you can see the values and pointer lines being dynamically updated in the Data Browser. To return execution to break level 1, select the **Step** button to continue single stepping until execution moves past the closing brace (}) of **store_shape()** and returns to the calling function.

Alternatively, rather than stepping, you can resume execution to the previous break level by selecting the **Continue** button.

Returning to a previous break level

When execution returns from **store_shape()**, execution is at line 17 in **do_bounce**, and the Workspace is back at break level 1. This means that you are back at the execution path that you left when you called **store_shape()** from the Workspace. You can now resume your original path of execution. However, since you changed global data by adding a new node to the linked list, you are resuming execution with a different program state.

Modifying the program

Now that you have a sense of the calling structure and flow of control for the Bounce program, you can make a modification that will integrate the function `draw_circle()` in the new module `circle.o`.

Calling your editor from the Workspace

To extend the Bounce program with a call to `draw_circle()`, you can change the code on line 16 of `do_bounce` to call `draw_circle()` instead of `draw_rectangle()`. To modify the code, do the following:

- 1 With `do_bounce` listed in the Source area, move the mouse pointer over number 12 in the line number column to the left of the code and press the Right mouse button.
- 2 In the pop-up menu, select **Edit line 12**.
- 3 In your editor, change lines 12 and 13 to replace the string `rectangle` with the string `circle`. The `if` statement will then look like the following:

```
if (strcmp(shape, "circle") == 0) )  
  
    draw_circle(col_table[ INDEX(count) - DECR ],  
               row_table[ INDEX(count) - DECR ] ;  
else  
  
    draw_rectangle(col_table[ INDEX(count) - DECR ],  
                  row_table[ INDEX(count) - DECR ] ;
```

- 4 In your editor, save the file.

Rebuilding your project

Since you have changed a source file, you need to bring these changes into CodeCenter by rebuilding your project.

In the middle of the Main Window, select the **Build** button.

Running the modified program with a command-line argument

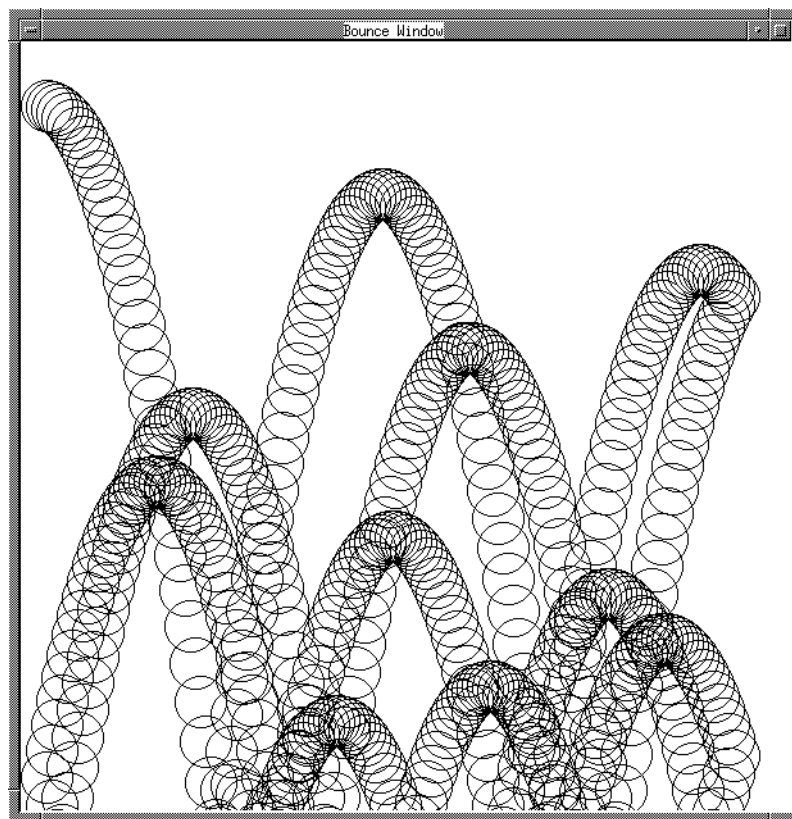
To have the modified Bounce program bounce a circle rather than a rectangle, you can delete the breakpoints and run the program using `circle` as a command-line argument. To delete the current breakpoints, do the following:

- 1 In the Main Window, display the **Debug** menu and select **Delete**.
- 2 In the Delete submenu, select **Delete All Debugging Items**.

To run the Bounce program, do the following:

- 1 At the bottom of the Project Browser, select the **Run** button.
- 2 At the input line of the dialog box, enter the string **circle**.
- 3 At the bottom of the dialog box, select the **Run** button.

You have now finished the enhancement, and the Bounce program successfully bounces a circle.



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 CodeCenter without using a makefile, go on to 'Using the load command' on page 57. To try loading your own code into CodeCenter, go to 'Loading your application' on page 67.

Leaving CodeCenter

If you want to quit CodeCenter now, save your project in case you want to revisit the Bounce program for further exploration on your own. To leave CodeCenter, do the following:

- 1 In the Main Window, display the CodeCenter menu and select **Quit** CodeCenter.
- 2 In the Quit Verification dialog box, select the **Save Project** button.
- 3 In the Save Project dialog box, check that the **Save to File** input line contains the name you want to use for this project file.
- 4 At the bottom of the dialog box, select the **Save Project** button.
- 5 In the Quit Verification dialog box, select the **Quit** button.

Chapter 4 Using the load command

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

Starting Chapter 4

To begin this chapter:

- 1 If CodeCenter is already running, exit your session as described in 'Quitting CodeCenter' on page 30.
- 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 CodeCenter:

```
% cd ~/ctutor_dir
% codecenter
```

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 **ctutor_dir** directory already contains object files for the source files used in this chapter, CodeCenter loads the object files. For example:

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

If the object files don't exist in **ctutor_dir**, CodeCenter creates them. For example:

```
1 -> load shapes.o
Cannot open '/net/test/ctutor_dir/shapes.o'.
No such file or directory
Executing:/net/.../cc /net/test/ctutor_dir/shapes.c
/net/test/ctutor_dir/shapes.c:
Loading: shapes.o
```

Load errors

If CodeCenter finds errors when you load a file, it immediately unloads it and reports errors in the Error Browser. For example, there are two versions of **bounce.c** in the tutorial directory. If you try to load both of them into the environment, you see output like this:

```
6 -> load bounce.c.bad
Loading: -I/usr/include/X11R4 -I/usr/include/X11R5
-I/usr/openwin/include bounce.c.bad
7 -> load bounce.c
Loading: -I/usr/include/X11R4 -I/usr/include/X11R5
-I/usr/openwin/include bounce.c
Unloading: bounce.c
*** Check error browser for more details. ***
Warning: 1 module currently not loaded.
```

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 CodeCenter 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, CodeCenter may not find the correct header files or libraries when loading.

1 Set the **program_name** option:

```
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 **\$(CFLAGS)** 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:

```
2-> setopt load_flags -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:

```
2-> setopt load_flags -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 CodeCenter commands and windowing options. They're discussed briefly on page 95, and in more detail in the options entry in the *CodeCenter 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, CodeCenter does not use most of the switches you specify until the file needs to be recompiled or you swap it to source. If CodeCenter cannot find an object file specified with the **load** command or the file is out of date relative to its source file, CodeCenter attempts to rebuild the file.

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

```
3 -> load shape.o
Cannot open 'shape.o'
No such file or directory
Executing: make shape.o
cc -g ... shape.c
Loading: shape.o
```

In this example, a makefile exists in the same directory as the object file, so CodeCenter 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, CodeCenter would have invoked the compiler directly.

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

```
4 -> load table.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 CodeCenter. You also attach the X11 library with the **-lX11** switch.

Load the **main.o**, **wait.o**, and **x.o** files and attach the **X11** library:

```
5-> load -G main.o wait.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 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 `bounce.c` as a source file:

```
6 -> load bounce.c
```

Linking from libraries

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

```
7 -> link
```

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

```
Undefined symbols:
extern void draw_rectangle();
```

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

```
8 -> load rectangle.o
Loading: rectangle.o
9 -> link
```

NOTE If you are using code that you changed as described in 'Modifying the program' on page 53, **draw_circle()** will be undefined and you should load **circle.o** instead of **rectangle.o**.

The Bounce program is now loaded into the CodeCenter 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 *CodeCenter Reference*.

Part II: Getting started with your own code

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

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

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

Chapter 5 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 CodeCenter 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 71).
- If your application uses a simple makefile, add CenterLine targets to it and use the CodeCenter **make** command to load your application (page 73).
- If your application uses complex makefiles, use the EZSTART utility to create a new makefile with CenterLine targets, and then use the CodeCenter **make** command to build the targets (page 77).

Performance issues

As mentioned in '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.

Using options to set load switches

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

The **sys_load_flags** option specifies 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**.

Setting system-wide load options

CodeCenter presets the **sys_load_flags** option to search for files. CodeCenter always appends the contents of this option to all **load** commands.

The value of the system-wide option varies with each platform. To verify that your **sys_load_flags** option contains the correct values for your platform, enter this command:

```
1 -> printopt sys_load_flags
```

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_flags** option, change the value of the option using the **setopt** command:

```
1 -> setopt sys_load_flags 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 command adds **-DBeta** to the list of switches:

```
1 -> setopt sys_load_flags $sys_load_flags -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:

```
1 -> setopt load_flags -DBETA -DDEBUG
2 -> load xyz.c
Loading: -DBETA -DDEBUG -L/usr/include xyz.c
```

In this example, the **-L/usr/include** switch was defined in **sys_load_flags** 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** switch 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:

```
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:

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

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

```
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.

```
1 -> ls *.o
abc.o xyz.o
2 -> setopt load_flags -DDEBUG
3 -> load -lm abc.o
Attaching: /usr/lib/libm.so
Loading: abc.o
4 -> load -G xyz.o
Loading: xyz.o
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 **-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 CodeCenter (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 CodeCenter **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 CodeCenter 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 two targets. The first is a standard C target.

- The first target (**all**) creates an executable named **my_program** from the three files **a.o**, **b.o**, and **c.o**, which are specified in **\$(OBJS)**
- The second target (**ccenter_obj**), a CL target, loads the objects specified in **\$(OBJS)** (**a.o**, **b.o**, and **c.o**) into ObjectCenter with the switches specified in **\$(CFLAGS)**.

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

CC = clcc
SRCS = a.c b.c c.c
OBJS = a.o b.o c.o
CFLAGS = -g -DDEBUG

# Standard targets
# The following target builds the application
# "my_program"

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

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

# The following target loads object files into
# CodeCenter, 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
ccenter_obj:
    echo "Starting a CL obj target"
    #setopt program_name my_program
    #load $(CFLAGS) $(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 CodeCenter **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 CodeCenter **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 107 for more information about options.)

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>#<CodeCenter command>
```

Before CodeCenter 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 CodeCenter 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.

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

In the example on page 74, note that the same set of switches used in the standard target, **\$(CFLAGS)**, is also included in the **load** command for the **ccenter_obj** target. In this example, **\$(CFLAGS)** 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 CodeCenter make command

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

When you issue the **make** Workspace command on a standard target, CodeCenter invokes the UNIX **make** utility to build the target. CodeCenter 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 CodeCenter.

When you issue **make** on a CL target, the CodeCenter 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:

```
20 -> make CL_target...
```

If you encounter problems

If you have problems designing your CL targets, refer to the **make** entry in the *CodeCenter Reference*. If you have trouble using the **load** and **link** commands, refer to the **load** entry in the *CodeCenter Reference* or 'Troubleshooting the load and link commands' on page 80.

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 CodeCenter.

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 CodeCenter **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 CodeCenter.

With **Makefile.cline**, you can use the CodeCenter **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 CodeCenter 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 CodeCenter 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 CodeCenter 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 CodeCenter 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 CodeCenter session, you can load the project file and quickly recapture the state of your work.

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

```
56 -> save filename
```

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

Loading your project file

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

- Specify the file when starting CodeCenter:

```
% codecenter ccenter.proj
```
- Load the project file from the Workspace:

```
1 -> load ccenter.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, CodeCenter 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 CodeCenter. 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 1 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	<p>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 and sys_load_flags options.</p> <p>Set the -I and -L switches to appropriate values. CodeCenter searches libraries in the order indicated in the load entry in the Manual Browser.</p> <p>You may need to unload the library and reload it for the load command to recognize the new flags.</p>
Unresolved references after link	<p>Use the unres command to get the list of unresolved references.</p> <p>Use the link -list command to check the library link order.</p> <p>Try issuing the link command a second time. Since CodeCenter's linker makes only one pass, a second link command may resolve the references.</p> <p>Issue the contents command in the Workspace to verify you have the correct libraries loaded.</p> <p>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. Modify your load switches or load_flags option appropriately.</p>

Table 1 Resolving **load** and **link** Problems (Continued)

load or link Problem	Possible Solutions
	<p>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:</p> <pre>1 -> unload <i>file</i></pre>
	<p>Reload the files with the load command and relink the project with the link command.</p>
Source module unloaded by CodeCenter	<p>This means an error exists in your source file. Check the Error Browser to understand the nature of the error.</p> <p>Make sure that you have included the right switches to locate header files by examining the load commands echoed to the Workspace.</p> <p>Invoke the editor if you need to correct a problem in your code.</p> <p>Explicitly unload the file in which the violations occurred by issuing the unload command in the Workspace.</p> <pre>1 -> unload <i>file</i></pre>
	<p>Load the file again and relink your application.</p>
Object module unloaded by CodeCenter	<p>If CodeCenter has trouble reading the debugging information, you should load your file with the -G switch.</p>
Cannot locate header file	<p>Make sure that the correct -I flags are being included from the load_flags and sys_load_flags options.</p>
Cannot load library file	<p>Make sure that the correct -L flags are being included from the load_flags and sys_load_flags options.</p>

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:

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


Chapter 6 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 CodeCenter, 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:

```
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 CodeCenter, 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:

```
3 -> setopt program_name test
```

The Run Window

CodeCenter 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 CodeCenter, 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:

```
3 -> unsetopt win_io
```

Detecting memory leaks

You can identify potential memory 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, CodeCenter 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?

CodeCenter'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.

CodeCenter does run-time error checking on both source and object code. Table 2 compares the kinds of run-time checking done for source files, instrumented object files, and object files (instrumented object files are explained on page 87).

Table 2 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. CodeCenter 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 CodeCenter 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. CodeCenter 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 CodeCenter'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 CodeCenter 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 92 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 CodeCenter:

- 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 90. This enables CodeCenter to perform the most run-time error checking possible on your code.
- 2 Run your application again to allow CodeCenter to locate the exact location of the problem.
- 3 Use the various debugging techniques described on page 92 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 CodeCenter 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, CodeCenter 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, CodeCenter does not report run-time warnings on dynamic type mismatches, dynamic used-before-set, or corrupted values, and it disables watchpoints on variables. CodeCenter continues to check pointer bounds.
- 4 Set **unset_value** to **0**. With this setting, CodeCenter 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, CodeCenter unloads the object file and loads the corresponding source file. After swapping your object file to source, you should select the **Run** button (or issue the **run** command) so CodeCenter can pinpoint the exact location of your run-time problem.

Setting the swap_uses_path option

When you swap an object file, CodeCenter looks 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, CodeCenter 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:

```
24 -> load test.c
Loading : /s3/beth/src/test.c
25 -> swap test.c
Unloading: /s3/beth/src/test.c
Loading: /s3/beth/obj/test.o
26 -> swap test.o
Unloading: /s3/beth/obj/test.o
Loading: /s3/beth/src/test.c
```

**Troubleshooting
the swap
command**

Table 3 contains information on possible solutions to swapping problems.

Table 3 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. CodeCenter 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 CodeCenter.
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

CodeCenter provides full interactive debugging for C. 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 CodeCenter 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 CodeCenter commands. When you enter code into the Workspace, CodeCenter parses your code like the C compiler.

The following examples use the Bounce program:

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

```
(break 1) 22 -> what is count
auto int count; /*Defined in `do_bounce`; currently
active */
```

- Use the Workspace to define a variable at a breakpoint. The following example defines a variable named **my_count**.

```
(break 1) 23 -> count;
(int) 300
(break 1) 24 -> int my_count = 1;
```

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

```
(break 1) 25 -> draw(200, 300);
(void)
```

- 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 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.

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 CodeCenter. The **action** command lets you insert C code or CodeCenter 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.

```
12 -> int count;  
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 valid 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 CodeCenter Workspace commands, such as **stop**, have a C function equivalent that you can call from 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 CodeCenter 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 CodeCenter command **stop**.

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

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**.

```
1 -> int *ptr;
2 -> int many[10];
3 -> ptr = &many[4];
(int *) 0x270790 /* many[4] */
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, CodeCenter 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**:

```
1 -> int abc;
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:

```
1 -> int *ptr;
2 -> ptr = (int *) malloc(20) ;
   (int *) 0x179d8c /* (allocated) */
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 Cross-Reference and Project Browsers are not available to aid you in visualizing functions and files in your application.

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

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

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

Debugging multiple processes

CodeCenter 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, CodeCenter 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 *CodeCenter Reference*. Support for debugging threaded applications is not available on all platforms. See the “Product limitations” section in the online *CodeCenter Reference* for details.

Using Ascii CodeCenter

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

- To run CodeCenter on a nongraphical workstation or over a dialup line.
- To gain faster startup time or to reduce the amount of memory needed to run CodeCenter.
- 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 CodeCenter with the **-ascii** switch:

```
% codecenter -ascii
```

Rebuilding your project

After you try the 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. CodeCenter 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, CodeCenter issues a **make** command. If there is no makefile in the directory containing the source file, CodeCenter directly invokes the C compiler.

If the object file is loaded with debugging information, CodeCenter 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 CodeCenter 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 CodeCenter 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 CodeCenter for prototyping new applications and as a test harness.

Using the Workspace for interactive prototyping

In component debugging mode, in addition to handling CodeCenter commands, the Workspace functions as a direct interface to CodeCenter's interpreter. Because the interpreter implements 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 to have CodeCenter execute it immediately. Also, when you execute code from the Workspace, CodeCenter'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 and **config_parser** entries in the *Reference*.

Because the Workspace allows you to enter any C statement and immediately execute it, the Workspace supports development through interactive prototyping. You can create code fragments or define variables, functions, and data structures as you go. For example, to define a function that adds two integers, you could enter:

```
29 -> int add(int x, int y)
30 +> {return x+y;};
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 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 program fragment in the Workspace. You can create stubs for external functions called by the code, and then execute your code to test it. 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 are visible at global scope in the Workspace. As a result, you may have to make static functions externally visible in order to use the Workspace sources as a separate file.

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

```
30 -> history #> ccenter_log_name
```

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

```
31 -> source command_file
```

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

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. CodeCenter's interpreter enables you to take this approach.

Using your editor and CodeCenter'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 CodeCenter

You can load an incomplete program into CodeCenter. 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 into CodeCenter 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

CodeCenter 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.

```

9 -> int seed;
10 -> seed = 7;
11 -> whatis seed
extern int seed; /* defined */
12 -> seed;
(int) 7
13 -> reinit
14 -> seed;
(int) 0
15 -> seed = 7;
(int) 7
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 4 Troubleshooting Run-Time Issues

Problem and Issues	Possible Solutions
Application working outside environment does not work inside the environment	<p>If you have run-time errors and warnings in your program, please refer to 'Responding to run-time problems' on page 88 to resolve run-time issues.</p> <p>If you have a working application and you just want to get your application working in CodeCenter, you should refer to the information in 'Dealing with large numbers of run-time warnings' on page 89 for ways you can scale down the amount of run-time error checking performed by CodeCenter.</p> <p>If your application uses the name of the application during execution, you must set the program_name option.</p>
Not enough virtual space	<p>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.</p> <p>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.</p> <p>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.</p>

Table 4 Troubleshooting Run-Time Issues

Problem and Issues	Possible Solutions
External preprocessors	<p>CodeCenter supports working directly with input files that are run through preprocessors that generate C files with the #line directives pointing back to the input file. Such preprocessors include YACC, certain SQL preprocessors, and preprocessors supporting parameterized types.</p> <p>See the preprocessed code entry in the <i>Reference</i> for complete information on this topic.</p>
Cannot declare functions in the Workspace	<p>If you declare a function in the Workspace, you must include the return type as part of the declaration. For example:</p> <pre data-bbox="651 670 1062 751"> 6 -> void func1() { 7 +> printf("Hello world\n"); 8 +> }</pre>

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:

```
7 -> contents #> filename
```


Chapter 7 Setting up your environment

CodeCenter 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 CodeCenter 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, CodeCenter 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. CodeCenter 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 CodeCenter 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.

```
1 -> setopt path /usr/prog/test
2 -> load abc.c def.o
Loading: /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 the cc_prog option

By default, CodeCenter uses **cc** as the C compiler. If you use a different compiler, such as **clcc** (CenterLine-C, CenterLine's optimizing ANSI C compiler) or **gcc**, set the **cc_prog** option. The **cc_prog** option affects the compiler used by the **make** and **load** Workspace commands. To set **cc_prog** from the Options Browser:

- 1 Select the text field next to the **cc_prog** option.
- 2 Enter the name of your compiler.

To find out if the CenterLine-C compiler is supplied with your platform, refer to your platform guide, which is available as an appendix to the online *Reference*. To use CenterLine-C, set **cc_prog** to **clcc**.

If desired, you can configure the parsing rules used by the interpreter when CodeCenter loads files as source. For further information, see the **config_parser** entry in the *Reference*.

Setting other options

Before you load your code, you may want to set the **sys_load_flags** and **load_flags** options as described on page 69. Table 5 outlines additional options that you may need to set depending on your application.

Table 5 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.
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 <i>Reference</i> .
Uses 8-bit character sets.	In the misc Option Set, set eight_bit to True .	Treats input and output as 8-bit characters.

Saving your option settings

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

- Specify them in your CodeCenter startup file.
- Save a project file and reload it in the next session. See the section 'Saving your project' on page 78 for more information.

Startup files in component debugging mode

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

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

Here is a sample **.ccenterinit** 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 CodeCenter first looks in the current working directory for **.ccenterinit**, you can have different **.ccenterinit** 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 **.pdmnit** startup file, first in the current working directory, and then in your home directory. It does not read the global **ccenterinit** 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 CodeCenter by using X resources, custom buttons, or the Workspace.

Creating custom commands

To create a custom command:

- 1 Display the **CodeCenter** menu, slide off on the **User Defined** submenu, and select **Add/Change/Delete**. CodeCenter 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.

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

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

Setting X resources

CodeCenter 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 CodeCenter.

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

Revision control system support

If you use either the **rscs** 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 **rscs** or **sccs**:

```
CodeCenter*ProjectBrowser.RevisionControl:
```

For more information about revision control system support, refer to the “Revision control systems” section in the *CodeCenter Reference*.

Integrating revision control by creating C functions

CodeCenter 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:

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

CodeCenter 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 CodeCenter under the control of FSF GNU Emacs. Refer to the emacs integration entry in the *CodeCenter Reference*.

Symbols

(pound sign), in CL targets 75
 +> prompt 93

A

actions, saving 78
Add Files menu item 34
 addresses, getting information on 95
 ANSI C
 additional documentation on 100
 setting options for 111
 array elements, displaying 48
 Ascii CodeCenter, using 97

B

batch_load option 98
 break levels
 and execution paths 46
 establishing 42, 87
 generating 95
 returning to previous 52
 breakpoints
 conditional 95
 deleting 53
 saving 78
 setting 41
 setting in parent and child 97
build command 99
Build menu item 25
 building, your project 99
 buttons, creating 113

C

C compilers
 ANSI option 111
 choosing 110

C functions, creating 114
 C interpreter, using 100
 C library functions, and run-time error checking 87
 calling structure, displaying 38
cc_prog option 110
ccenter.proj file 78
.ccenterinit file 112
 CenterLine target, *See* CL targets
centerline_* functions 95
 child processes, setting breakpoints 97
 CL targets
 additional documentation on 73
 building 76
 designing 73
 and EZSTART 77
 loading your application with 73Ð76
 for maintaining projects 79
clxterm client 85
 code fragments, entering in the Workspace 93
 CodeCenter
 loading your application 67Ð81
 quitting 30
 running your application 83Ð105
 setting up your environment 107Ð115
 commands
 build 25, 99
 cont 88
 continue 42, 52
 creating custom 113
 debug 6
 for setting options 109
 history 101
 in CL targets 73
 info 95
 instrument 18
 make 15, 76, 77
 next 44
 print 24
 run 85
 save 78
 setopt 70

- sh** 75
- source** 102
- start** 103
- step** 43, 52
- stop on** 96
- swap** 90
- unload** 99
- unres** 80
- whatis** 23
- conditional breakpoints 95
- Contents window, components of 35
- continuing, execution 42, 88
- corefiles, debugging 3D15
- correcting, run-time errors 13D30
- creating
 - buttons 113
 - CL targets 75
 - commands 113
 - functions 114
- Cross-Reference Browser 36
- custom commands, creating 113
- customizing, your environment 112

D

- data
 - examining 47
 - printing the value of 94
 - visualizing 93
- Data Browser
 - components of 9
 - contents of 47
 - dereferencing pointers in 9
 - dynamic update of data items 52
 - moving data items in 51
 - saving space in 49
 - using 93
 - zooming 48
- data items, moving 51
- <data> return type 36
- databases, setting option for preprocessors 111

- debug** command 6
- debugging
 - corefiles 3D15
 - fully linked executables 97
 - items, deleting 53
 - modes, switching 5
 - multiple processes 97
 - techniques 92D98
 - threaded processes 97
- deleting breakpoints 53
- dependencies, in CL targets 73, 75
- dereferencing, pointers in the Data Browser 9
- designing, CL targets 73
- display, setting your iv
- displaying, functions, variables, headers, types,
 - and typedefs 35
- dragging, data items 51

E

- Edit Line menu item 8
- editing
 - search path used when 109
 - source code 25, 29
- editor, specifying iv
- eight_bit** option 111
- elements of array, displaying 48
- enhancing
 - programs 31D55
 - your application 100
- environment variables
 - DISPLAY iv
 - EDITOR iv
- environment, setting up 107D115
- Error Browser 87
 - contents of 27
 - using to suppress warnings 89
- errors
 - large numbers of 89
 - responding to 88
 - run-time 13D30, 86

- saving to file 81
- examining
 - data 47
 - variables with the Workspace 93
- executables, debugging 97
- executing
 - main()** 85
 - your program with different test data 103
- execution
 - continuing 42, 88
 - interrupting 96
 - single stepping 43
- execution paths, and break levels 46
- expressions, displaying 23, 93
- <extern> return type 36
- EZSTART
 - loading your application with 77

F

- files
 - ccenter.proj** 78
 - loading 16
 - reloading 29
 - swapping 90
- folder symbol in Data Browser 48
- fork** system call 97
- fully linked executables, debugging 97
- functions
 - calling from Workspace 45
 - CenterLine 95
 - creating 114
 - displaying 35
 - listing 40
 - printing the number of calls to 94
 - resolving undefined 102
 - stepping through 44

G

- G** switch, for loading files 62
- g** switch, overriding 62
- global variables, initializing 103

H

- header files
 - displaying 35
 - resolving problems locating 81
 - search path for 61
- history** command 101

I

- I** switch, for loading files 61
- identifiers, displaying the definition of 23
- incremental linking 99
- info** command 95
- instrumented object files
 - run-time error checking 86
 - speed of execution 87
 - swapping 90
- instrumenting object files 18, 34
- interpreter, using 100

L

- L** switch, for loading files 61
- l** switch
 - and **load_flags** option 69
 - for loading libraries 62
- leaks, finding memory 85
- libraries
 - attaching 62
 - linking from 64
 - resolving problems in loading 81
 - search path for 61
- link** command 80

link -list command 80
 linked lists, examining 47
 linking

- from libraries 64
- incrementally 99

 listing, search path used when 109
load command 80
load_flags option 61
 loading

- additional documentation on 64
- files 16
- object files that don't exist 62
- object files using CL targets 75
- object files with debugging information 62
- object files without debugging information 62
- project files 79
- search path used when 109
- setting load switches with options 70
- source files 63
- switches 61
- the Bounce program 59D64
- with CL targets 73D76
- with EZSTART 77
- with the **load** command 71D72
- your application 67D81

 load-time error checking 27

M

m4 preprocessor, setting options for 111
 macros, in CL targets 76
main() function, executing 85
make command 15, 76, 77
 makefiles

- and EZSTART 77
- and loading your application 69
- and reloading files 99
- loading files with 15
- maintaining 79

 memory leaks, finding 85

memory locations, monitoring 96
 messages, saving to file 81
 metacharacter expansion, preventing 75
 moving

- among break levels 52
- data items 51

 multiple processes, debugging 97

N

name, of program, setting 85

O

object files

- and watchpoints 96
- instrumenting 18
- loading 16
- reloading 99
- run-time error checking 87
- swapping 90
- that don't exist 62
- with debugging information 62, 71, 75
- without debugging information 62, 71, 75

 options

- ansi** 111
- batch_load** 98
- cc_prog** 110
- eight_bit** 111
- for setting load switches 70
- load_flags** 61
- path** 109
- preprocessor** 111
- program_name** 111
- save_memory** 89
- saving values 78
- setting 107D115
- setting in CL targets 75
- swap_uses_path** 90
- unset_value** 89

 Options Browser 109

P

- parent processes, setting breakpoints 97
- parsing rules, configuring 110
- path used for loading, listing, editing, swapping 109
- pdm
 - additional documentation on 97
 - setting executable name 111
 - switching modes 5
- performance issues 69
- pointers, dereferencing in the Data Browser 9
- preprocessors, setting option for 111
- print** command 24
- printing
 - number of calls to a function 94
 - the value of data structures 94
- program_name** option 85, 111
- programs
 - enhancing 31Ð55
 - running 7
- Project Browser
 - components of 16
 - exporting contents to a file 105
 - instrumenting files 34
 - setting options 109
- project file
 - definition 30, 78
 - for saving options 112
 - loading 79
 - saving 78
- projects
 - and CL targets 79
 - establishing 15
 - saving 30, 78
- Properties windows 109
- prototyping, in the Workspace 100

Q

- quitting, CodeCenter 30

R

- rcs**
 - creating functions for 114
 - setting resources for 114
- rebuilding 11
 - object files 62
 - your program 25
 - your project 99
- regular object files
 - run-time error checking 86
 - swapping 90
- reloading files 99
- reloading, files 29
- Restart Session menu item 15
- return type, of functions 36
- revision control systems
 - creating functions for 114
 - setting X resources for 114
- run** command 85
- Run Window 85
- running
 - programs 7
 - test suites 103
 - your application 83Ð105
- run-time error checking
 - discussion 86
 - enabling 18
 - reducing 89
- Runtime Error dialog box 20
- run-time errors, correcting 13Ð30
- run-time problems, responding to 88

S

- save** command 78
- save_memory** option 89
- saving
 - error messages to file 81
 - projects 30, 78
 - your option settings 112

sccs

- creating functions for 114
- setting resources for 114

segmentation faults 5

setopt command 70

setting breakpoints 41

sh command 75

shell commands

- creating 113
- in CL targets 73, 75

shrinking, data items 49

signals, saving 78

single stepping 43

Source area

- execution symbol 7
- lines causing warnings or errors 28
- listing a file containing a particular function 40
- listing the corresponding source file 87
- pop-up menu 8

source command 102

source files

- editing 25
- loading 16, 63
- reloading 99
- run-time error checking 86
- swapping 90

standard targets, building 76

start command 103

startup files

- .ccenterinit** 112
- for saving options 112

stepping, execution 43

stop on command 96

suppressing, warnings 88

suppressions, saving 78

swap command 88, 90, 91

swap_uses_path option 90

swapping

- files 90
- search path used when 109

switches, for loading 61

switching, debugging modes 5

system testing 103

T

tabs, in CL targets 75

targets

- CL, *See also* CL targets
- standard 76

terminal emulators, and custom commands 113

test suites, running 103

testing

- automating 98
- running test suites 103
- system testing 103
- unit testing 102
- your application 100

<text> return type 36

threaded processes, debugging 97

troubleshooting

- load** and **link** commands 80
- run-time issues 104
- the **swap** command 91

types and typedefs, displaying 35

U

undefined symbols, listing 80

unit testing 102

unload command 99

unres command 80

unset_value option 89

using, Ascii CodeCenter 97

V

variables

- displaying the value of 24
- examining with the Workspace 93

W

warnings

- ignoring 88
- large numbers of 89
- responding to 88
- run-time 86
- saving suppressions 78
- saving to file 81
- suppressing 88

watchpoints 96

win_io option 85

Workspace

- additional documentation on 93
- break levels 46
- calling functions 45
- commands, *See* commands
- directing output to 85
- entering code fragments 93
- pdm prompt 5
- prototyping in 100
- setting options in 109–110
- unit testing in 102
- using 92

X

X resources, setting 114

xterm client, and Run window 85

Z

zooming, in Data Browser 48