ObjectCenter Reference

Version 2.1.1

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

Distribution

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

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

A version of FSF GNU Emacs compatible with the CenterLine Emacs Main Window is also available on the same host. For more information, please refer to the **README** file in the following directory:

/pub/TOOLS/emacs

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

What this manual is about

This manual is a complete reference to Version 2.1.1 of ObjectCenter™. This alphabetical reference contains entries for topics as well as for Workspace commands and predefined functions. Some examples of topics are as follows: ANSI C, built-in functions, commands, debugging, language selection, options, environment variables, C library functions, templates, and X resources.

Each entry in the *Reference* that describes an ObjectCenter command has a quick reference check-off box at the top showing the command modes in which the command is available: component debugging mode (**cdm**), process debugging mode (**pdm**), or both.

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

What you should know before starting

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

Moreover, we assume that readers of the *Reference* are already familiar with ObjectCenter by having read the *ObjectCenter Tutorial* and / or the *ObjectCenter User's Guide*.

For more information

The *Reference* does not contain extensive information about using ObjectCenter's graphical user interface; see the *ObjectCenter User's Guide* for this information. The *ObjectCenter User's Guide* provides a task-based look at ObjectCenter; it explains how to use the graphical user interface to load, manage, run, and debug programs within ObjectCenter.

We designed the *ObjectCenter Tutorial* as a hands-on introduction to ObjectCenter. It leads you step by step through an ObjectCenter session, using either the Motif or the OPEN LOOK interface.

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

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

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

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

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

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

Documentation conventions

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

literal names	Bold words or characters in command descriptions represent words or values that you must use literally.
user-supplied values	Italic words or characters in command descriptions represent values that you must supply. Italic words in text also indicate the first use of a new term, or emphasis.
`sample user input`	In interactive examples, information that you must enter appears in **`this typeface.`**
`output/source code`	Information that the system displays appears in `this typeface.`
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
<<*none*>>	In a "Description" section, indicates how a command performs with no arguments.

Contents

List of Tables

List of Figures

List of Tips

Alphabetical Reference

action

sets a debugging action

cdm	pdm
✔	

Command syntax

action

action [at] *"file":line*

action [at] *line*

action [in] *function*

action [on] *address*

action [on] *lvalue*

action [on] *variable*

Description

<< *none* >>	Executes the defined action at every executable line of loaded source code. This does not apply to statements executed directly in the Workspace or statements in object code.
[**at**] "*file*":*line*	Executes the defined action when program execution reaches the specified line in the specified file.
[**at**] *line*	Executes the defined action when program execution reaches the specified line in the current file.
[**in**] *function*	Executes the defined action whenever the specified function is entered.
[**on**] *address*	Executes the defined action whenever the byte at the specified address is modified, except for statements in object code. The *address* argument must be a hexadecimal value.

[**on**] *lvalue*	Executes the defined action whenever the referenced address, such as a dereferenced pointer, is modified.
[**on**] *variable*	Executes the defined action whenever the specified variable is modified.

Options

The following ObjectCenter options affect the **action** command:

list_action	(Ascii ObjectCenter only) Displays actions that execute everywhere when listing the source line at which they were triggered.
save_memory	Actions cannot be set on dynamic memory if **save_memory** is set.

See the **options** entry for more details about each option.

Usage

Use the **action** command to specify a debugging action, written in C++ code, that is executed when program execution reaches a specified location or changes a specified value. The **action** command allows you to customize and extend ObjectCenter's built-in debugging facilities. For example, using **action** you can design conditional breakpoints. Actions can be listed with the **status** command and deleted with the **delete** command.

NOTE Use the **when** command instead of **action** when you are in process debugging mode.

Triggering actions from the Workspace or from actions

A function defined in the Workspace that changes the value of the target triggers the associated action in the same way that a function in your program would. For example, in the following sequence the call to **set_x()** triggers the action set on x:

```
(break 1) 150 -> int x;
(int) 0
(break 1) 151 -> load_header iostream.h
Loading (C++): -I. /tmp/OC.afd/iostream.h
(break 1) 152 -> action on x
Enter body of action. Use braces when entering
multiple statements.
action -> cout << "triggered on x" << endl;
action (2) set on address 0x16db48.
(break 1) 153 -> void set_x() { x = 6; }
(break 1) 154 -> set_x();
triggered on x
(void)
```

NOTE The output from the **cout** statement in the action appears in the window where you started ObjectCenter.

However, actions are *not* triggered by statements under the following conditions:

- Actions are not triggered when the value of the target is changed by an immediate statement in the Workspace. For example, in the following sequence the statement **x = 5** does *not* trigger an action on **x:**

```
(break 1) -> action on x
Enter body of action. Use braces when entering
multiple statements.
action -> printf( "triggered on x\n");
action (2) set on address 0x16db48.
(break 1) -> x = 5;
(int) 5
```

- Actions are not triggered when the value of the target is changed within an action itself; that is, actions are not recursive.

  For example, in the following sequence, although the call to **set_x2()** does trigger an action on **x**, incrementing **x** within the action does not trigger a second action:

```
(break 1) -> action on x
Enter body of action. Use braces when entering
multiple statements.
action -> {
action +> printf("triggered on x\n");
action +> ++x;
action +> }
action (2) set on address 0x16db48.
(break 1) -> void set_x2() { x = 7;
(break 1) +> printf("in setx_2()\n");}
(break 1) -> set_x2();
(void)
```

  Given the preceding actions and function call, the following appears in the Run window:

```
triggered on x
 in setx_2()
```

Setting actions on object code

In addition to defining actions on source code, you can define actions on code that is loaded in object form. If the object code contains debugging information from the compiler (that is, if the object code was compiled using the **-g** switch *and* was loaded into ObjectCenter without the **-G** switch), then you can set an action at a line, at a line in a specified file, or in a function. Actions *cannot* be set on an address, lvalue, or variable in object code.

If the object code does not contain debugging information (either the object code was compiled without the **-g** switch or was loaded into ObjectCenter with the **-G** switch), then actions can be set only on a function name.

Blocks

Each debugging action consists of one or more C++ or C statements. If the action comprises more than one statement, use braces to make the action a single block of C++ or C code.

Variables and parameters

A debugging action can use any variables that are in scope at the location where the action is set.

Formal parameters and automatic variables may be used only if the action is set at a specific location within a file, as opposed to being set on a variable or an address.

print command

You cannot use the ObjectCenter **print** command in an action; instead, use either the **cout** or the **printf()** function.

Setting watchpoints

The following action, set on line 10 of **main.c**, will print the value of **total** when execution reaches that line. If **total** is 0, then execution is halted by a call to **centerline_stop()**, a function that is equivalent to the ObjectCenter **stop** command issued without arguments.

```
-> action at 10
Setting action at "main.c":10, main()
Enter body of action. Use braces for multiple
statements.
action -> {
action +> printf("total = %d\n", total);
action +> if (total == 0) centerline_stop("");
action +> }
action (1) set at "main.c":10, test().
->
```

Note that braces make the multi-statement action a single block.

NOTE When you save your project to a project file, actions may not be saved in the form in which you entered them. For example, if you set an action on a function, the action is set on the file and line number at which the function occurs rather than on the function name. As a result, actions may not behave in the way you expect them to when you reload your project.

Restrictions

Actions set on addresses that are modified while executing in object code are not performed.

See Also

built-in functions, **delete**, **status**, **stop**, **when**

alias

creates an alias for a command

cdm	pdm
✔	✔

alias

alias *name*

alias *name text*

alias *name text alias_args*

Description

<< *none* >>	Lists all aliases currently set.
name	Lists the text value for the specified alias *name.*
name text	Sets the *name* string to the value of the *text* string.
name text alias_args	Sets the *name* string to the value of the *text* string and defines arguments for the alias. (cdm only)

Usage

Use the **alias** command to create an alternative name for ObjectCenter commands. When an alias is detected at the beginning of a command line, its text is used in place of the name. Use aliases to create shortcuts for frequently used commands.

Default aliases

In addition to aliases that you can create, ObjectCenter comes with several default aliases, such as **ls** and **pwd**. When you issue the **alias** command without arguments, the default aliases are displayed along with any that you have defined. For example:

```
-> alias s step
-> alias
ls              sh ls
pwd             sh pwd
assign          print
set             print
s               step
```

NOTE To save an alias permanently, place its definition in your **.ocenterinit** file.

Alias argument symbols

To specify arguments for an alias, use the following symbols in the definition of the alias:

#: *n*	The *n*th argument on the command line. Arguments are numbered starting with 0, which is the alias name.
#: ^	The first argument on the command line—same as **#:1**, the argument that follows the alias name.
#: * *or* **#***	All arguments on the command line except the 0 argument, the alias name itself.
#: $	Last argument on the command line.
#$	Same as **#: $** unless it matches one of the patterns listed next.
#$*identifier*	Substitutes the value of the ObjectCenter option, if one exists, with the specified name; otherwise, substitutes the value of the named environment variable. For example, **#$path** substitutes the value of the ObjectCenter **path** option, if it is set. Similarly, **#$HOME** substitutes the current value of the **HOME** environment variable.
#$*environ_var*	Substitutes the value of the named environment variable. For example, including **#$HOME** substitutes the current value of the **HOME** environment variable.
#${*option*}	Substitutes the named ObjectCenter option value. For example, **#${load_flags}** substitutes the loading switches that you have set in ObjectCenter.

Examples

The following examples demonstrate how to define and use aliases that take arguments.

NOTE If you are defining an alias in the Workspace and the alias takes arguments, escape the # character with a backslash (\) so that ObjectCenter does not expand the variable before recording the definition. However, do not use a backslash to escape the # character in alias definitions in your **.ocenterinit** file.

The following alias lists the file **hello.c** in your home directory. **#$HOME** expands to the directory set by the **HOME** environment variable.

```
-> alias l list \#$HOME/hello.c
-> l
Warning: this file is not loaded.
     1: #include <stdio.h>
     2:
     3: main()
     4:
     5: {
     6:         printf("Hello world\n");
     7: }
```

You can redefine the alias to list a specified C source file in your home directory. **#:1.c** expands to the first argument on the command line.

```
-> alias l list \#$HOME/\#:1.c
-> l hello
```

The alias in the next example adds one or more directories to ObjectCenter's search path. **#${path}** expands to the current value of ObjectCenter's **path** option, and **#*** expands to all arguments but the **alias** name. In this example we assume that the **path** option is unset. We define the **addpath** alias, and then use it to add first one directory, and then two more, to the **path** option.

```
-> alias addpath setopt path \#${path} \#*
-> addpath ~/c_programs
-> printopt path
path ~/c_programs
string - list of directories to search for source,
object, and library files
-> addpath ~/ctutor_dir ~/tctutor_dir
-> printopt path
path ~/c_programs ~/ctutor_dir ~/tctutor_dir
string - list of directories to search for source,
object, and library files
```

Use the following form (that is, without the backslashes) to add these aliases to your **.ocenterinit** file.

```
alias l list #$HOME/#:1.c
alias addpath setopt path #${path} #*
```

Restrictions

You cannot use the following form in process debugging mode:

alias *name text alias_args*

In process debugging mode, the alias command cannot evaluate another alias. That is, given this syntax:

alias *name text*

the *text* string cannot include the name of another alias.

See Also

keybind, **unalias**

ANSI C

ObjectCenter supports both Kernighan and Ritchie (K&R) C and the ANSI standard C language. The default setting is K&R C.

NOTE The information in this entry about ANSI C applies when you set the **ansi** option and also do any of the following:

- Use C mode
- Load files with the **-C** switch
- Set the **primary_language** option to C

See the **language selection** entry on page 175 for additional information.

See the **config_c_parser** entry on page 103 for more information about configuring ObjectCenter to emulate a particular C compiler.

In the rest of this entry, we describe the following topics:

- Using the ANSI mode of ObjectCenter
- ANSI conventions always in effect
- Known incompatibilities and bugs in ObjectCenter's ANSI support
- Using function prototypes, including generating them with the **proto** command and loading them from libraries

Using ANSI

To work with ANSI C code using ObjectCenter's C mode, use the **setopt** command to set ObjectCenter's **primary_language** and **ansi** options:

```
setopt primary_language C
setopt ansi
```

With these options set, ObjectCenter loads and runs C code strictly according to the ANSI standard.

If you wish to use ANSI C rules while you are in C++ mode, then leave the **primary_language** option set to the default, which is C++ for ObjectCenter, and just use the **ansi** option. In C++ mode, the **ansi** option affects only the preprocessor.

As shown in the following example, ObjectCenter in C mode with **ansi** set accepts constructs not found in K&R C, but found in ANSI C.

```
C++ 3 -> cmode
C Workspace Enabled.
C 4 -> const int j=5;
Error #733: 'const' is undefined.
C 5 -> setopt ansi
C 6 -> const int j=5;
c 7 -> j;
(int const) 5
```

NOTE If you are using ANSI, be sure to read the "Specifying the search path for loading libraries and #include files" **TIP** on page 196.

TIP: When does the ansi option take effect?

If you forget to set the **ansi** option when you load an ANSI C source file, you'll get errors for code that is ANSI-compliant and not K&R C. To fix this problem, you must not only set the **ansi** option and load the file, you must also explicitly unload the file with the **unload** command before you load the file.

Here's an example. Suppose your source file named **ansi.c** contains the following code:

```
main()
 {
 const int i =4;
 }
```

Attempting to load this file generates an error:

```
C++ 26 -> load -C ansi.c
 Loading (C): ansi.c
 Unloading: ansi.c
 Warning: 1 module currently not loaded.
```

The error message is as follows:

```
Line: 3 E#733 'const' is undefined
```

Now you realize you forgot to set the **ansi** option, which you do, but instead of unloading and loading, you simply load; as a result you get the same error again:

```
C++ 27 -> setopt ansi
C++ 28 -> load -C ansi.c
Loading (C): ansi.c
Unloading: ansi.c
Warning: 1 module currently not loaded.
```

The correct way to cause the **ansi** option to take effect is to explicitly unload and then load:

```
C++ 29 -> unload ansi.c
C++ 30 -> load -C ansi.c
Loading (C): ansi.c
```

The **ansi** option works in a way that's similar to the way the **load_flags** option works; see "When does the load_flags option have precedence?" **TIP** on page 191 for more information.

ANSI conventions always in effect

The following ANSI features are always in effect in ObjectCenter in C mode, even if **ansi** is not set:

- ANSI C function prototypes are always accepted by ObjectCenter; however, in K&R mode they do not force type coercion. Compare the following results involving the coercion of an **int** to a **double**:

```
C++ 10 -> cmode
C Workspace Enabled.
C 11 -> unsetopt ansi
C 12 -> load -lm
Attaching: /usr/lib/libm.a
C 13 -> double sqrt(double);
C 14 -> sqrt(3);
Warning #69: Serious type mismatch in call to
function 'sqrt':
Argument #1 has type (int) but type (double) was
expected.
Defined/declared in "workspace":13
Linking from '/usr/lib/libm.a' .... Linking
completed.
Linking from '/usr/lib/libc.sa.1.6' ... Linking
completed.
(double) 2.523368e-157
C 15 -> setopt ansi
C 16 -> sqrt(3);
double) 1.732051e+00
```

- In ObjectCenter, preprocessor directives do not have to start with the first character of a line. They can begin anywhere on a line, but the **#** character that begins the directive must be the first non-whitespace character.
- ObjectCenter ignores **#pragma** directives in K&R mode, as well as in ANSI mode.
- The unsigned-suffix is allowed even in K&R mode:

```
-> unsetopt ansi
-> unsigned int u = 5u;
-> u;
 (unsigned int) 0x5
```

- In accordance with the ANSI standard, ObjectCenter concatenates adjacent string literals.

- ObjectCenter places labels and variables into separate name spaces. This means that a label and a variable with the same name can be visible at the same time.
- Union initialization lists are always allowed.

Known ANSI incompatibilities and bugs

This section lists the known incompatibilities and bugs in ObjectCenter's support of ANSI C. If you discover other problems with the ANSI support, please contact CenterLine Software (email address: **objectcenter-support@centerline.com**).

In preparing this list, we assume that all language-related ObjectCenter options are set to C, and that the **ansi** option is set.

Libraries and header files

ObjectCenter loads whatever libraries and **#include** files you indicate for it to load—whether or not they are ANSI-compliant and whether or not you are in **ansi** mode.

NOTE If you are using an ANSI C compiler, see the "Specifying the search path for loading libraries and #include files" **TIP** on page 196.

Function prototypes

In function prototypes with multiple sets of parentheses, only one set can contain parameter types if you are using the **void** keyword. For example, the following function prototypes should work in ObjectCenter but they do not:

```
-> int (*g(void)) (int);
Error #905: The function parameter list has an
illegal format.
-> int (*g(void)) (int k);
Error #905: The function parameter list has an
illegal format.
```

The workaround is to use one of the following forms:

```
-> int (*g()) (int k);
-> int (*g(void)) ();
```

Scoping rules

ObjectCenter gives a within-block **extern** declaration file scope. ANSI gives it block scope.

International features

ObjectCenter recognizes wide character constants and wide string literals, but it treats them as normal character and string constants. Trigraphs are not implemented.

Using function prototypes

One of the big changes in ANSI C is the addition of function prototypes, which you can use to ensure that functions are being called with the proper arguments and that return values are being used properly.

Generating function prototypes

ObjectCenter can automatically generate function prototypes for your loaded functions, which can be helpful when you are migrating K&R C applications to ANSI C. To generate prototypes, load your code in source form, then issue the **proto** command:

```
-> load -C const.c
-> proto const.c
Writing prototypes to a file. Output file name?
const.proto
```

You can load prototype files just like C source files. To avoid redefinition errors, load them before the corresponding source files.

Type coercion

You can load function prototypes in K&R mode and in ANSI mode. The only difference concerns type coercion of function arguments.

In K&R mode, arguments are not coerced, they are only checked; this means that warning messages might be generated upon a type mismatch. Prototypes in K&R mode do not affect the meaning of your program; they only provide extra checking.

In ANSI mode, arguments are coerced; however, following ANSI specifications, a prototype loaded in one module does not cause argument coercion in another module. This is a natural consequence of the C language's "separate compilation" model.

assign

assigns a value to a variable

cdm	pdm
✔	✔

Command syntax **assign** *variable = expression*

Description

variable = expression Evaluates an *expression* (second argument) and assigns the value of the expression to a *variable* (first argument).

Usage

Use the **assign** command to evaluate an expression and assign its value to a variable. Assigning a value to a variable in the Workspace allows you to either directly manipulate values in code that you are debugging or to set values for code you are creating in the Workspace. The **assign** and **set** commands are functionally identical.

Direct evaluation

You can also assign a value to a variable without using **assign** (or **set**), simply by evaluating an assignment expression in the Workspace. For example:

```
-> int i;
-> assign i = 2
(int) 2
-> i = 5;
(int) 5
```

See Also **print**, **set**

attach

attaches to a running process

cdm	pdm
	✔

Command syntax **attach** *process_id*

Description

process_id Attaches ObjectCenter to the running process identified by *process_id*. The process can be running outside or inside ObjectCenter. You can attach to only one process at a time.

Usage

When you attach to a running process, ObjectCenter stops the process. You can then examine and modify the process with any ObjectCenter commands that are available in process debugging mode. If you want the process to continue running, use the **cont** command. Use the **detach** command to release a process from ObjectCenter's control. If you try to attach a process while you are already attached to another process, ObjectCenter prompts you to detach before attaching.

You can use the **attach** command in combination with debug to attach an executable file to an already running process. That is, you can use the following two commands:

```
(pdm) 1 -> debug my_a.out
(pdm) 2 -> attach my_process_id
```

instead of the following:

```
(pdm) 2 -> debug my_a.out my_process_id
```

NOTE If you leave process debugging mode or use the **run** command while you have an attached process, you kill that process.

See Also **debug**, **destruct**, **pdm**, **run**

browse_base

displays base classes in the Workspace

cdm	pdm
✔	

Command syntax **browse_base** *class_name*

Description

class_name Ascii ObjectCenter: Displays all base classes for *class_name*. Shows whether *class_name* was derived publicly or privately.

Switches

-a Ascii ObjectCenter: Displays the complete inheritance hierarchy for the specified class item.

Usage

Use the **browse_base** command to examine information about base classes or about the complete inheritance hierarchy for a given class.

When displaying inherited base classes in the Workspace, **browse_base** uses indentation to indicate inheritance levels. For example, the following output from **browse_base** with the **-a** switch shows that the class **Car** is derived publicly from the class **Land_vehicle**, which in turn is derived publicly from the class **Vehicle**.

```
-> browse_base -a Car
Base Classes:
Land_vehicle <Public>
Vehicle <Public>
```

See Also **browse_class**, **browse_data_members**, **browse_derived**, **browse_friends**, **browse_member_functions**, **classinfo**, **list_classes**

browse_class

displays base and derived classes and members in the Workspace

cdm	pdm
✔	

Command syntax

browse_class *class_name*

Description

class_name Ascii ObjectCenter: Lists the base classes and derived classes for *class_name* in the Workspace.

Displays data members and member functions that are accessible to *class_name*.

Lists the members' access level (public, protected, or private).

Switches

-l Ascii ObjectCenter: Limits the display of members to those defined in the specified class, not listing inherited members.

Options

The following ObjectCenter option affects the **browse_class command:**

show_inheritance Displays class members with the full inheritance path showing how members are inherited. If unset, shows a truncated inheritance path giving only the defining class.

See the options entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **browse_class** command to list in the Workspace the full range of inheritance information for a particular class. The information is as follows: base classes, derived classes, data members, and member functions.

For example, the following output from **browse_class** shows that the class **Land_vehicle** publicly derives from a base class **Vehicle** and in turn serves as a base class for the publicly derived classes Car and Truck.

The data members **doors**, **mileage**, and **model** and the functions **calcMileage()**, **setModel()**, **show()**, and **setMileage()**, together with two constructors and one destructor, are all local to the class **Car**. The data member **tires** and two constructor functions are local to **Land_vehicle**, while the other data members and member functions derive from the class **Vehicle**.

```
-> browse_class Car
 Base Classes:
 Land_vehicle <Public>

 Derived Classes:
 (nothing)
 Data Member Interface:
 <Protected> int doors
 <Protected> float mileage
 <Protected> char *model
 <Protected> int Land_vehicle::tires
 <Protected> char *Land_vehicle:Vehicle::serial_num
 <Protected> char *Land_vehicle:Vehicle::owner
 <Protected> char *Land_vehicle:Vehicle::city
 <Protected> int Land_vehicle:Vehicle::passengers
 Member Function Interface:
 <Public> Car::Car(char *sn = "XXXXX", char *own = "Unknown", int dr = 4)
 <Public> inline Car::Car(const class Car &)
 <Public> Car::~Car()
 <Public> float Car::calcMileage()
 <Public> void Car::setModel(char *mdl)
 <Public> virtual void Car::show(ostream &strm = <const value>)
 <Protected> inline void Car::setMileage(float mpg)
 <Public> Land_vehicle::Land_vehicle(char *sn = "XXXXX", char *own =
"Unknown", int tr = 4)
 <Public> inline Land_vehicle::Land_vehicle(const class Land_vehicle &)
 <Public> Land_vehicle:Vehicle::Vehicle(char *sn = "XXXXX", char *own =
"Unknown")
 <Public> inline Land_vehicle:Vehicle::Vehicle(const class Vehicle &)
 <Public> Land_vehicle:Vehicle::~Vehicle()
 <Public> void Land_vehicle:Vehicle::setSerial(char *sn)
 <Public> void Land_vehicle:Vehicle::setOwner(char *own)
 <Public> void Land_vehicle:Vehicle::setPassengers(int p)
 <Public> virtual void Land_vehicle:Vehicle::move(char *cty)
 <Public> virtual void Land_vehicle:Vehicle::show(ostream &strm = <const
value>)
```

Indicating inheritance

When ObjectCenter shows the inheritance path, a single colon (**:**) indicates inheritance and the scoping operator (**::**) indicates the class that actually defines a member. The way that ObjectCenter shows inheritance is affected by the current value of the **show_inheritance** option.

In the previous example, to indicate that **setOwner()** is a member function that the class **Land_vehicle** inherits publicly from its base class Vehicle, the **browse_class** command displays the following line:

```
<Public> void Land_vehicle:Vehicle::setOwner(char *own)
```

See Also

browse_base, **browse_data_members**, **browse_derived**, **browse_friends**, **browse_member_functions**, **list_classes**

browse_data_members

displays data members in a class in the Workspace

cdm	pdm
✔	

Command syntax

browse_data_members *class_name*

Description

class_name Ascii ObjectCenter: Displays the data members in the specified class and gives the members' access level (public, protected, or private).

Switches

-l Ascii ObjectCenter: Limits the display of members to those defined locally (actually defined in the specified class)

Options

The following ObjectCenter option affects the **browse_data_members** command:

show_inheritance Displays class members with the full inheritance path showing how members are inherited. If unset, shows a truncated inheritance path giving only the defining class.

See the options entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **browse_data_members** command to display information about data members for a given class.

Example

The following output from **browse_data_members** shows that the class **Car** defines the data members **doors**, **mileage**, and **model** locally, while it inherits tires from its base class **Land_vehicle** and derives still other data members from the class **Vehicle** through **Land_vehicle**.

```
-> browse_data_members Car
 Data Member Interface:
 <Protected> int doors
 <Protected> float mileage
 <Protected> char *model
 <Protected> int Land_vehicle::tires
 <Protected> char *Land_vehicle:Vehicle::serial_num
 <Protected> char *Land_vehicle:Vehicle::owner
 <Protected> char *Land_vehicle:Vehicle::city
 <Protected> int Land_vehicle:Vehicle::passengers
```

Indicating inheritance

When ObjectCenter shows the inheritance path, a single colon (:) indicates inheritance and the scoping operator (::) indicates the class that actually defines a member. The way that ObjectCenter shows inheritance is affected by the current value of the **show_inheritance** option.

In the previous example, to indicate that **passengers** is inherited by **Car** from the class **Vehicle** through **Land_vehicle**, the **browse_data_members** command displays the following line:

```
<Protected> int Land_vehicle:Vehicle::passengers
```

See Also

browse_base, **browse_class**, **browse_derived**, **browse_friends**, **browse_member_functions**, **classinfo**, **list_classes**

browse_derived

displays derived classes in the Workspace

cdm	pdm
✔	

Command syntax

browse_derived *class_name*

Description

class_name	Ascii ObjectCenter: Displays classes derived directly from the specified class and lists the form of derivation (public or private).

Switches

-a	Ascii ObjectCenter: Displays the complete inheritance hierarchy (shows the classes that are derived from classes derived from the specified class and so on, recursively).

Usage

Use the **browse_derived** command to examine information about derived classes or the complete inheritance hierarchy from one class.

When displaying derived classes, **browse_derived** uses indentation to indicate inheritance levels. For example, the following output from **browse_derived** with the **-a** switch shows that the classes **Land_vehicle** and **Water_vehicle** are derived from the class **Vehicle** and that the class **Car** is, in turn, derived from **Land_vehicle**.

```
-> browse_derived -a Vehicle
 Derived Classes:
 Land_vehicle <Public>
           Car <Public>
 Water_vehicle <Public>
```

See Also

browse_base, **browse_class**, **browse_data_members**, **browse_friends**, **browse_member_functions**, **classinfo**, **list_classes**

browse_friends

displays friend classes and friend functions in the Workspace

cdm	pdm
✔	

Command syntax **browse_friends** *class_name*

Description

class_name Ascii ObjectCenter: Displays friend classes and friend functions for the specified class.

Usage Use the **browse_friends** command to examine information about friend classes and friend functions.

Example

```
-> browse_friends String
Friends:
Class friends:
(nothing)
Function friends:
String operator +(String &str1, String &str2)
```

See Also **browse_base**, **browse_class**, **browse_data_members**, **browse_derived**, **browse_member_functions**, **classinfo**, **list_classes**

browse_member_functions

displays member functions in a class in the Workspace

cdm	pdm
✔	

Command syntax

browse_member_functions *class_name*

Description

class_name Ascii ObjectCenter: Displays the member functions in the specified class and gives the members' access level (public, protected, or private).

Switches

-l Ascii ObjectCenter: Limits the display of member functions to those defined locally (actually defined in the specified class).

The following option affects the **browse_member_functions command:**

show_inheritance Displays class members with the full inheritance path showing how members are inherited. If unset, shows a truncated inheritance path giving only the defining class.

See the options entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **browse_member_functions** command to display information about member functions in a given class.

Example

The following output from the **browse_member_functions** command shows that two constructor functions (**Car::Car()**), one destructor function (**Car::~Car()**), and the functions **calcMileage()**, **setModel()**, **show()**, and **setMileage()** are local to the class Car. The other member functions are derived either from the class **Land_vehicle** or the class Vehicle through **Land_vehicle**.

```
-> browse_member_functions Car
 Member Function Interface:
 <Public> Car::Car(char *sn = "XXXXX", char *own = "Unknown", int dr = 4)
 <Public> inline Car::Car(const class Car &)
 <Public> Car::~Car()
 <Public> float Car::calcMileage()
 <Public> void Car::setModel(char *mdl)
 <Public> virtual void Car::show(ostream &strm = <const value>)
 <Protected> inline void Car::setMileage(float mpg)
 <Public> Land_vehicle::Land_vehicle(char *sn = "XXXXX", char *own =
"Unknown", int tr = 4)
 <Public> inline Land_vehicle::Land_vehicle(const class Land_vehicle &)
 <Public> Land_vehicle:Vehicle::Vehicle(char *sn = "XXXXX", char *own =
"Unknown")
 <Public> inline Land_vehicle:Vehicle::Vehicle(const class Vehicle &)
 <Public> Land_vehicle:Vehicle::~Vehicle()
 <Public> void Land_vehicle:Vehicle::setSerial(char *sn)
 <Public> void Land_vehicle:Vehicle::setOwner(char *own)
 <Public> void Land_vehicle:Vehicle::setPassengers(int p)
 <Public> virtual void Land_vehicle:Vehicle::move(char *cty)
 <Public> virtual void Land_vehicle:Vehicle::show(ostream &strm = <const
value>)
```

Indicating inheritance

When ObjectCenter shows the inheritance path, a single colon (:) indicates inheritance and the scoping operator (::) indicates the class that actually defines a member. The way that ObjectCenter shows inheritance is affected by the current value of the **show_inheritance** option.

In the previous example, to indicate that **setOwner()** is a member function that the class **Car** inherits publicly from **Vehicle** through **Land_vehicle**, ObjectCenter displays the following:

```
<Public> void Land_vehicle:Vehicle::setOwner(char *own)
```

See Also

browse_base, **browse_class**, **browse_data_members**, **browse_derived**, **browse_friends**, **classinfo**, **list_classes**

build

reloads all files in the project that have changed

cdm	pdm
✔	✔

Command syntax **build**

Description

<< none >> Updates your project by looking at all the files currently loaded and reloading all files that have changed.

In process debugging mode, ObjectCenter reloads the executable (for instance, **a.out**) if the executable is newer than the current one.

Options

The following ObjectCenter options affect the **build** command:

auto_compile Automatically compiles missing or outdated object files. If you invoke a **build** from the Project Browser, this option is ignored; missing or out-of-date files are always recompiled.

ccargs Specifies arguments passed to **cc** when invoked from ObjectCenter.

cxxargs Specifies default arguments passed to the C++ translator when invoked by ObjectCenter.

c_suffixes Specifies file extensions to search for when ObjectCenter needs to find a C source file that corresponds to a given object file.

cxx_suffixes Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.

make_args Specifies the command-line arguments passed to the UNIX **make** command by ObjectCenter's **make** command.

make_hfiles Checks header files to determine whether a file should be reloaded. If you are loading a large project, setting this option can be time consuming.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **build** command to keep your project current when you are working with multiple files.

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

An object file that is older than its source counterpart is recompiled, then reloaded. If an object file has been loaded with debugging information (compiled with the **-g** switch) and if the **make_hfiles** option is set, ObjectCenter also 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.

Also, an object file is reloaded if the file has been recompiled since it was loaded.

Files not loaded

The **build** command also attempts to reload any files that failed to load previously because they contained an error. The **build** command attempts to reload such files each time it is issued until it successfully loads the file or until the file is explicitly unloaded using the unload command.

If a file fails to load because the **load** switches are incorrect, issuing **build** will not help, since build uses the same incorrect switches. In this case, you need to unload and reload the file, using the correct switches with load.

Recompiling

When a recompile is necessary, ObjectCenter first looks for a makefile in the source directory. If there is a makefile, ObjectCenter calls **make**, passing the value of the **make_args** option. If no makefile exists in the source directory, ObjectCenter invokes either the C++ translator or the

C compiler directly. If the loaded file that is motivating the recompile was loaded with the **-C** switch, then ObjectCenter calls **cc**, passing the value of the **ccargs** option; otherwise, ObjectCenter calls **CC**, passing the value of the **cxxargs** option.

See Also **debug**, **load**, **make**, **unload**

built-in comments

See Table 1 for a list of predefined comments that ObjectCenter recognizes and uses to suppress certain kinds of error checking. Use these comments in source code that would ordinarily cause a violation that you want to ignore.

Table 1 Predefined Comments Used to Suppress Load-Time Errors

Comment	What the Comment Tells ObjectCenter To Do
/*VARARGS*/	Allow the following function to take a variable number of arguments. If you are using the **varargs(3)** macro package you need not use this comment.
/*VARARGS*n***/**	Suppress reporting of a variable number of arguments, after *n* arguments.
/*NOTREACHED*/	Suppress warning that the following statement cannot be reached.
/*ARGSUSED*/	Suppress warning that formal parameters of the function are not used.
/*SUPPRESS *n***/**	Suppress reporting of violation #*n*. If this comment appears at the global level of a file, ObjectCenter suppresses the violation for the entire file. If the comment appears within a function, the violation is suppressed only for the following line. See the violations entry in the Manual Browser for a list of violations and their numbers.
/*EMPTY*/	Suppress reporting on empty bodies, such as in **if** statements and **for** loops. The /*EMPTY*/ comment must appear on its own line preceding the statement on which reporting is to be suppressed.

built-in functions

Each ObjectCenter command has a C++ function equivalent that can be used to call the command from C or C++ code. The names for these functions all begin with a **centerline_ prefix**. For example, you can call the **print** command in your C or C++ code by calling the function **centerline_print(**" "**)**. All such functions return an **int** value and take a **string** as an argument.

Setting watchpoints

The **centerline_stop(**" "**)** call is equivalent to issuing the **stop** command with no arguments. It is typically used to create a conditional debugging action that interrupts execution when a condition becomes true, as shown in the following example:

```
-> int i;
-> action
Enter body of action. Use braces when entering
multiple statements.
action -> if ( i == -1 ) centerline_stop("");
action #1 set.
-> status
 (1) action /* everywhere */
     1: if ( i == -1 ) centerline_stop("");
```

In addition to the function equivalents for commands, ObjectCenter provides the following predefined functions that you can use in your programs:

- **centerline_getopt()**
- **centerline_malloct()**
- **centerline_[open | get | next | close]_sym**
- **centerline_true()**
- **centerline_unset()**
- **centerline_untype()**

Because the ObjectCenter functions and command equivalents have C linkage, to call them within a C++ module you must declare them with the **extern** "**C**" label. For example:

```
extern "C" char *centerline_getopt(char *);
```

ObjectCenter functions return 0 upon success, except where the nature of the function requires a different return value scheme, such as with **centerline_getopt()**. If an error occurs during execution of an ObjectCenter function, a non-zero value is returned and a message is displayed in the Workspace or the Error Browser.

NOTE The **centerline_typeof** keyword, which is available in CodeCenter, is not supported in ObjectCenter.

See Also **centerline_getopt()**, **centerline_malloct()**, **centerline_[open | get | next | close]_sym** , **centerline_true()**, **centerline_unset()**, **centerline_untype()**

built-in macros

For your convenience, ObjectCenter predefines several macros, including __OBJECTCENTER__, OBJECTCENTER, and _ _CENTERLINE_ _ to the value **1**. You can use these macros to conditionalize your code so certain code is used only when you are working in ObjectCenter.

For example, your code would look like this:

```
< program code >
...
#ifdef __OBJECTCENTER__
< code to be run only when in ObjectCenter >
#endif
...
< more program code >
```

NOTE You can also use the **centerline_true()** built-in function to determine at run time if your program is running in ObjectCenter.

To allow conditional compilation for source files that are compiled by both the C++ translator and the C compiler, ObjectCenter predefines the macros **__cplusplus** and **c_plusplus**. These macros are predefined to the value **1**, the same way they are predefined by the C++ translator.

See Table 2 for a list of these and other macros recognized by ObjectCenter.

Table 2 Macros Recognized by ObjectCenter

Name of Macro	Macro Definition	Additional Information
__CENTERLINE__	Always defined as **1**.	None.
OBJECTCENTER	Always defined as **1**.	
__OBJECTCENTER__	Always defined as **1**.	
__FILE__	Name of the file being read.	Also predefined by **CC** and **cc**.

Table 2 Macros Recognized by ObjectCenter (Continued)

Name of Macro	Macro Definition	Additional Information
__FUNC__	Name of the function being read.	We do not recommend that you use this macro, since it is not available in other C++ or C implementations.
__LINE__	Line number of the file being read.	Also predefined by **CC** and **cc**.
__DATE__	Date the file was read (“*Mmm dd yyyy*”).	Defined only if the **ansi** option is set.
__TIME__	Time the file was read (“*hh:mm:ss*”).	Defined only if the **ansi** option is set.
__STDC__	Always defined as **1**.	Defined only if the **ansi** option is set. This macro is defined by C compilers and interpreters that conform to the ANSI standard.
__cplusplus	Always defined as **1**.	Also predefined by **CC.**
c_plusplus	Always defined as **1**.	Also predefined by **CC.**

NOTE The **c_plusplus** macro is included only for backward compatibility with AT&T C++ 1.2 source code. When writing new code, use the **__cplusplus** macro instead of **c_plusplus**.

See Also **built-in functions**

catch

traps signals before they reach the program

cdm	pdm
✔	✔

Command syntax

catch

catch *signal_name*

catch *signal_number*

Description

<< *none* >>	Lists the unprefixed names of the signals that are currently caught.
signal-name	Enables trapping for the designated signal and generates a break level whenever the signal is generated.
signal-number	Enables trapping for the designated signal and generates a break level whenever the signal is generated.

Usage

Use the **catch** command to trap signals before they reach the program; each signal is either caught or ignored by ObjectCenter. Once a signal is trapped, ObjectCenter generates a break level.

In component mode (cdm), when a signal is caught and a break level is generated, the signal is consumed. Ignoring the signal at the break level and continuing execution does not regenerate the signal and pass it to the program.

However, in process debugging mode (pdm), you can use the **cont** command to pass the signal number to your program.

Signal numbers

To obtain the number for a signal, consult the UNIX reference manuals for your system.

Signals caught

To view a list of the signals caught for your platform, use the **catch** command without any arguments.

Signal name

With the **catch** command, the signal name can be in uppercase or lowercase letters, and it can be used with or without the prefix “SIG”. For example, the following commands are equivalent:

```
-> catch SIGALRM
-> catch sigalrm
-> catch ALRM
-> catch alrm
```

Restrictions

Control-z at the command prompt is not interfered with (Ascii ObjectCenter only).

Control-z during execution or in the Run Window is always handled as a signal-deliver, generating an error if not trapped by the user program.

Ignoring SIGINT causes SIGQUIT to perform interruption duties. Ignoring both of them interferes with stopping execution.

The signals SIGTTIN and SIGTTOU will never suspend execution; if not trapped and ignored they will generate an error.

When an **exec()** is done within ObjectCenter, the inherited signal mask only includes signals that have been ignored; see the **ignore** command. Also, the SIGQUIT, SIGTRAP, and SIGEMT signals are never present in the inherited signal mask (cdm only).

See Also

cont, **ignore**

cc and other C compilers

This entry applies to the underlying compiler used by ObjectCenter. This entry does not apply to the type of C intermediate code (K&R versus ANSI C) that ObjectCenter generates to interpret C++ code internally. For information about the type of C intermediate code (K&R versus ANSI C) that ObjectCenter generates, see the **code generation** entry on page 91.

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

If you want to change the default setting, you can use the **ansi** option and/or the **config_c_parser** command to control the C language features supported by ObjectCenter. See the **ANSI C** entry on page 12 and the **config_c_parser** entry on page 103 for more information.

NOTE If you are using ANSI C and/or a compiler that uses "non-standard" libraries, be sure to read the "Specifying the search path for loading libraries and #include files" **TIP** on page 196.

You can use the **primary_language** option to set the language of the programming environment to be C by default, as shown:

```
-> setopt primary_language C
```

See the **language selection** entry on page 175 for more information about setting the default language for the environment to C instead of C++.

In general, ObjectCenter accepts exactly the same language accepted by the typical implementation of the **cc** command with a few exceptions:

Function prototypes always parsed

ObjectCenter always parses function prototypes, even when the **ansi** option is unset. This means that you cannot use names declared with **typedef** as formal parameters.

Typedef name as a formal parameter not allowed

For instance, the following construction is accepted by **cc** but not by ObjectCenter:

```
typedef int integer;
/* THIS IS NOT ACCEPTED BY OBJECTCENTER */
float convert(integer)
int integer;
{/* ... */}
```

When this sample code is loaded into ObjectCenter, ObjectCenter generates one of these errors: “Missing a parameter name”, “Prototype lacks parameters”, or “Illegal parameter list.”

Empty array brackets in structure not allowed

ObjectCenter does not permit empty array brackets in structure declarations:

```
struct open_ended {int first; float rest[];};
```

This construction is not legal C++ code, and ObjectCenter does not accept it. Instead, ObjectCenter generates the error “Structure member declarations require that all array dimensions be specified.” Some **cc** implementations accept such a structure declaration with only a warning.

NOTE See the *ObjectCenter Platform Guide* for any additional information about compatibility between ObjectCenter and the C compiler native to your platform.

Intentional bugs

There are several bugs in **cc** implementations that over the years have crept into a great deal of code and have become de facto features. We have reproduced three of these bugs in ObjectCenter to allow greater compatibility with existing code; for these bugs, ObjectCenter reproduces the compiler’s behavior described below.

The first two bugs involve **lvalues**, objects that may be assigned a value. Many compilers consider the result of a cast to be an **lvalue** if the type of the cast and the type of object are both integers or pointers of the same size.

```
int *p;
++(int)p; /* Adds 1 (not 4) to ‘p’ */
```

Nonetheless, according to the ANSI standard, the proper form of the previous example should be:

```
int *p;
p = (int *)((char *)p + 1);
      /* Adds 1 (not 4) to‘p’ */
```

Many compilers allow the result of a conditional expression to be used as an lvalue, if the expression being tested is a constant. For example:

```
int i, j;
((1 > 0) ? i : j) = 3;
      /* Assigns to ‘i’ (not ‘j’) */
```

Finally, many compilers allow the semicolon after the last field declaration in a tag definition to be omitted. For example:

```
struct s
 {
    int i;
    double d /* Missing last ‘;’ */
 };
```

ObjectCenter emulates the compiler’s behavior described in the examples for these three bugs. In addition, ObjectCenter generates warnings when it detects any of these bugs; however, by default the warnings are suppressed.

NOTE For a complete list of ObjectCenter diagnostic messages, including those suppressed by default, use the Manual Browser to view the “violations” topic; you can invoke the Manual Browser by issuing the command **man violations** in the Workspace.

To unsuppress the warnings, use the **unsuppress** command; see the **unsuppress** entry on page 402 for more information.

clcc

The CenterLine-C compiler (invoked with **clcc**) is a CenterLine product independent of ObjectCenter. If it is installed on your workstation, you can invoke the CenterLine-C compiler from within or outside of the ObjectCenter environment.

See the *CenterLine-C Programmer’s Guide* for more information about the CenterLine-C compiler.

gcc

On some platforms ObjectCenter allows you to load object files compiled with **gcc** using the **-g** compiler switch. See the *ObjectCenter Platform Guide* for more information about using **gcc** on your particular platform.

See Also

alias, **ANSI C**, **CC**, **config_c_parser**, **options**

CC

shell command to invoke ObjectCenter's C++ translator

Command syntax

CC [*switch] ... filename ...*

See Table 3 on page 46 for a list of the switches you can use. You can also find this information by typing the following command at the shell prompt:

```
$ man CC
```

Description

CC is ObjectCenter's version of the AT&T C++ Language system, including the translator (**cfront**), which translates C++ code to C code, prior to compilation. The command uses **clpp** for preprocessing, **cfront** for syntax and type checking, and **cc** for code generation.

CC takes arguments ending in **.c**, **.C**, **.cpp**, **.cxx**, **.cc**, or **.i** to be C++ source files. The **.i** files are presumed to be the output of the preprocessor. Both **.s** and **.o** files are also accepted by the **CC** command and passed to **cc**.

For each C++ source file, **CC** creates a temporary file, *file***.c**, in **/usr/tmp**, containing the generated C file for compilation with cc. Use the **-F -.**suffix switch to save a copy of this file in the current directory with the name file.*suffix*. The **+i** option saves a copy of the generated C code (minus **#line** directives) in the current directory with the name *file***..c**.

Version of C++ supported

ObjectCenter's translator is compatible with the C++ translator as defined by Release 3.0 of the AT&T C++ Language System. For information about the definition of the C++ language supported by ObjectCenter, see the *AT&T C++ Language System Release 3.0 Product Reference Manual*, which is supplied with ObjectCenter. For information about compatibility with previous releases of the translator and future compatibility, see the USL documentation in the Manual Browser.

Advantages of using ObjectCenter's C++

Using the C++ translator supplied with ObjectCenter instead of another C++ translator offers the following advantages:

- Object files compiled using ObjectCenter's translator contain more debugging information, allowing improved debugging of object code inside of ObjectCenter.
- ObjectCenter's translator places header-file dependency information in object files. This may not happen with other C++ translators.

  This means that if a header file is changed and you issue **build**, affected object files will be automatically recompiled if they were initially compiled using ObjectCenter's translator. If you had used another C++ translator, they might not be recompiled.
- ObjectCenter's translator places more reliable line number information in object files.
- ObjectCenter supports a facility that helps you avoid unnecessary recompilation of common header files, which can save significant compilation time in certain situations. See the for more information about using this feature of ObjectCenter's C++ language system**.**
- ObjectCenter allows you to specify demand-driven code generation with the **-dd=on** and **-dd=off** switches; the default setting is **-dd=on**, except for header files. Using demand-driven code generation reduces compilation time.

  You can also use the **-dd=on** switch with the **load** command, and get the benefit of demand-driven code generation with source files loaded into ObjectCenter.

  See the **demand-driven code generation** entry on page 129 for more information.

NOTE ObjectCenter might not support the C++ translator native to your platform; see your *ObjectCenter Platform Guide*. If your native C++ translator is not supported and you want to load object files into ObjectCenter, you must use ObjectCenter's version of **CC** to create the object files.

Switches

Table 3 describes the switches to the **CC** command.

NOTE In addition to the switches in Table 3, **CC** accepts other switches and passes them on to the C compilation system tools. See the UNIX manual pages for **clpp** for preprocessor switches, the **cc** or **clcc** manual page for C compiler switches, and **ld** manual page for link editor switches.

Table 3 CC Command-Line Switches

Name of Switch	What The Switch Tells CC to Do
-C	Do not discard comments; pass them through to the output file.
-dd=[on | off]	Use demand-driven code generation exclusively (**-dd=on**); this is the default setting. See the "demand-driven code generation" section on page 129 for more information.
-dryrun	Show but do not execute the commands constructed by the compilation driver.
-ec *string*	Pass *string* to the C compiler. Be sure to use double-quotes if necessary to pass spaces or other characters significant to the shell. For example, **-ec -fsingle** passes **-fsingle** to the C compiler.
-el *string*	Pass *string* to the linker. Be sure to use double-quotes if necessary to pass spaces or other characters significant to the shell. For example, **-el** "**-a archive**" passes **-a archive** to the linker.
-E	Run only the preprocessor on the C++ source files and send the result to standard output.
-F	Run only the preprocessor and **cfront** on the C++ source files, and send the result to standard output. The output contains **#line** directives.

Table 3 CC Command-Line Switches (Continued)

Name of Switch	What The Switch Tells CC to Do
-flags_cc=*string*	Pass *string* to the C compiler. The **-ec** switch now provides similar functionality. The **-flags_cc=***string* switch is provided for backwards compatibility with previous versions of the CenterLine-C++ compiler.
-flags_cpp=*string*	Pass *string* to the C preprocessor. For instance, if you want to use the pre-ANSI rather than the ANSI C preprocessor, use the following form of this switch: **-flags_cpp=-traditional**. Be sure to use double-quotes if necessary to pass spaces or other characters significant to the shell.
-g	Produce additional symbol table information for debugging purposes.
-hdrepos=*directory*	Use *directory* as a repository for precompiled header files, and look in *directory* for the *filename* (precompiled header information file) used with **+k[=***filename*]. See the **precompiled header files** entry on page 271 for more information.
-ispace	Causes less inlining by decreasing inline cutoff. This in general decreases program speed but makes the program smaller. Inlining of very small inline functions continues to be done.
-ispeed	Causes more inlining by increasing inline cutoff. This in general increases program speed at the expense of increased space.
-mt	Compiles and links for multithreaded code. Passes **-D_REENTRANT** to the preprocessor. Sets the header file include path to contain the thread safe header files. If the value of the LIB_ID environment variable is **C**, sets the value of LIB_ID to **C_mt** so that your code explicitly links in the thread safe version of the C++ library, **libC_mt.so**. Passes **-lthread** to the linker.

Table 3 CC Command-Line Switches (Continued)

Name of Switch	What The Switch Tells CC to Do
-ncksysincl	Do not check timestamps of files included with angle brackets (< >) when determining if a precompiled header file is out of date. See the **+k** switch (below) and also the **precompiled header files** entry on page 271 for more information.
-nCenterLine	Generate code without CenterLine extensions, including demand-driven code generation, CenterLine built-in functions, and CenterLine debugging information.
-pg	Enable profiling. When you use the **-pg** switch, **CC** sets the value of the LIB_ID environment variable to **C_p** so that your code explicitly links to the profiling version of the C++ library, which is named **libC_p.a**. See the "Using gprof" section on page 51 for more information. NOTE: **CC** does not change the value of LIB_ID if it has been explicitly set by the user.
-pta, -ptd*pathname* **-ptf, -pth, -pti** **-ptk, -ptm***pathname* **-ptn, -pto***pathname* **-ptr***pathname***, -ptt** **-pts, -ptv**	These switches affect the template instantiation process. See Table 30 on page 360 for more information about these particular switches, and see the **templates** entry on page 335 for more information about templates generally.
-set_lib_id=*value*	Set the value of the LIB_ID environment variable to value. See Table 4 on page 52 for more information about LIB_ID.
-.*suffix*	When used in combination with **-E** or **-F**, place the output from each input file in a file with the specified suffix in the current directory.
-v	Verbose mode. Print the command line for each process as it begins to execute.
-Yp,*pathname*	Use *pathname* as the location of the C preprocessor. This switch overrides the value of the cppC environment variable. The default value of cppC is **$CCROOTDIR/clpp**.

Table 3 CC Command-Line Switches (Continued)

Name of Switch	What The Switch Tells CC to Do
+a[0|1]	The C++ compiler can generate either ANSI C or K&R C declarations. The **+a** switch specifies which style of declarations to produce. The default, **+a0**, causes the compiler to produce K&R C-style declarations. The **+a1** switch causes the compiler to produce ANSI C-conforming declarations. This switch also affects the library path unless you have set the environment variable CCLIBDIR to a directory other than the default value set at installation. If the CCLIBDIR value is different from the default, the library search path is the value to which you have set CCLIBDIR. If the CCLIBDIR value is the default value, the search path contains the K&R C libraries if you specify **+a0** and the ANSI C libraries if you specify **+a1**. The **clpp** ANSI C preprocessor provides ANSI preprocessing features whether or not you use the **+a** switch. For more information about K&R versus ANSI C intermediate code generation, see the **code generation** entry on page 91.
+d	Do not inline-expand functions declared inline.
+e[0|1]	Only to be used on classes for which virtual functions are present, and all the virtual functions are either inline or pure. In this circumstance, this switch optimizes a program to use less space by ensuring that only one virtual table per class is generated. Specifically, **+e1** causes virtual tables to be external and defined. The **+e0** switch causes virtual tables to be external but only declared. **CC** ignores this switch for any class that contains an out-of-line virtual function.
+i	Leave the intermediate **..c** files in the current directory during the compilation process (note that there are two dots before the c suffix). These files do not contain any preprocessing directives, although the files passed to the C compiler do. When templates are used, it causes the instantiation system to leave **..c** files in the template repository (by default, **ptrepository**).

Table 3 CC Command-Line Switches (Continued)

Name of Switch	What The Switch Tells CC to Do
+k[=*filename***]**	Save and restore header files from a repository; if *filename* is provided, use it to determine which header files to save and restore. By default this switch is not set, meaning do not save and restore header files from the repository. See the **-ncksysincl** switch elsewhere in this table, and also see the **precompiled header files** entry on page 271 for more information.
+p	Disallow all anachronistic constructs. Ordinarily the translator warns about anachronistic constructs; under **+p** (for "pure"), the translator will not compile code containing anachronistic constructs. See the *AT&T C++ Language System Product Reference Manual* for a list of anachronisms.
+V	Cause calls to operator **new** to behave as in standard versions of **cfront** 3.0. This is the default behavior unless you compile with **-g**. Note however that if you specify **-g** (without **+V**) CC generates calls to **centerline_new** and / or **centerline_vec_new** to enable additional run-time error checking. These calls will generate errors if your code is not linked with the CenterLine C++ library. Use **+V** when you specify **-g** if you must link your code with other C++ libraries or if you plan to export library code to other users.
+w	Warn about constructs that are likely to be mistakes, be nonportable, or be inefficient. Without the **+w** switch, the compiler issues warnings only about constructs that are almost certainly errors.
+x *file*	Read a file of size and alignments created by compiling and executing **szal.c**. The form of the created file is identical to the entries in **size.h**. This option is useful for cross compilations and for porting the translator.

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

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

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

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

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

Using gprof

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

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

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

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

   ```
   % myexec
   ```

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

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

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

Environment variables used by CC

The **CC** script uses environment variables to locate files it needs to run and for other environmental information. You can override the values of these environment variables by setting them to different locations. For example, if you are using a different C compiler, you could issue this command (from the C-shell):

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

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

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

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

See Table 4 for a list of the variables used by **CC.**

Table 4 Environment Variables Used by **CC**

Name of Environment Variable	Default Value	Meaning
AON	**+a0**	K&R **(+a0)** or ANSI**(+a1)** C style declarations.
CLCCDIR	<set via install>	Directory containing **clcc**.
CCLIBDIR	<set via install>	Directory containing C++ libraries. By default, ObjectCenter always places the path specified in CCLIBDIR in front of any directories specified with **-L**.
CCROOTDIR	<set via install>	Directory containing **cfront**, **c++filt**, **CC**, **patch**, **ptcomp**, **ptlink**, etc.
CENTERLINE_CC_VERBOSE	1	Displays messages to aid in setting the ccC environment variable correctly.
CL_REPOS_LOCK_MAX_WAIT	**7200**	Total number of seconds to wait for a precompiled header file lock.

Table 4 Environment Variables Used by **CC** (Continued)

Name of Environment Variable	Default Value	Meaning
CL_REPOS_LOCK_STALE_TIME	**1440**	Minutes since last modification time of a precompiled header file lock before it is deleted.
CLcleanR	**$CCROOTDIR/skip/cleanr**	Precompiled header repository cleanup.
CPLUS	**-Dc_plusplus=1**	1.2 **cpp** C++ constant for backward compatibility.
CPPFLAGS	Platform-specific flags, including **-Amachine -C -lang-c++ -DCENTERLINE_CLPP=1**	Flags to the preprocessor. NOTE: -DCENTERLINE_CLPP=1 is undefined if you override the value of the cppC environment variable.
DEMANGLE	**1**	1 enables C++ link-time error message demangling.
FS	**0**	1 if **-fs** switch is available.
I	<set via install>	Directory for C++ include files.
LIBRARY	**-l$LIB_ID**	Standard C++ library name.
LIB_ID	**C**	Modify **LIBRARY**; the full path will be **$CCLIBDIR/lib${LIB_ID}.a.**
LINE_OPT	<unset>	Set to "**+L**" to generate source line number information using the format "**#line %d**" instead of "**# %d**"
LOPT	**-L**	**cc** switch for linker library directory.

Table 4 Environment Variables Used by **CC** (Continued)

Name of Environment Variable	Default Value	Meaning
LPPEXPAND	"**-l++**"	Specifies the string to which the command line argument "**-l++**" expands.
NM	**nm**	Location of **nm**.
NMFLAGS	unset on most platforms	Extra switches for **nm**.
PTHDR	**.H**, **.h**, **.HH**, **.hh**, **.HXX**, **.hxx**, **.hpp**	List of header file suffixes **ptlink** uses to look up template type declarations.
PTSRC	**.C**, **.c**, **.CC**, **.cc**, **.CXX**, **.cxx**, **.cpp**	List of source file suffixes **ptlink** uses to look up template type definitions.
PTOPTS	unset	Default switches to be passed to the template instantiation system.
TMPDIR	**/usr/tmp**	Directory used as root of temporary file directory for C++ compilation.
ccC	**$CLCCDIR/clcc -w**	The C compiler (the value of ccC defaults to the native C compiler if **clcc** is not available).
cfrontC	**$CCROOTDIR/cfront**	The C++ translator.
cPLUS	**-D__cplusplus=1**	2.0 **cpp** C++ constant for ANSI C conformance.
cplusfiltC	**$CCROOTDIR/c++filt**	C++ link error message filter.
cppC	**$CCROOTDIR/clpp**	The C preprocessor.
munchC	**$CCROOTDIR/munch**	The **munch** executable.
patchC	**$CCROOTDIR/patch**	The **patch** executable.
ptcompC	**$CCROOTDIR/ptcomp**	The **ptcomp** executable.

Table 4 Environment Variables Used by **CC** (Continued)

Name of Environment Variable	Default Value	Meaning
ptlinkC	**$CCROOTDIR/ptlink**	The **ptlink** executable.
skippp	**$CCROOTDIR/skippp**	The precompiled header preprocessor.

See Also **demand-driven code generation**, **language selection**, **precompiled header files**, **templates**

cd

changes the current working directory

cdm	pdm
✔	✔

Command syntax

cd

cd *pathname*

Description

<< *none* >>	Changes the working directory for ObjectCenter to your home directory.
pathname	Changes the working directory for ObjectCenter to the designated *pathname*. UNIX wildcards are allowed.

Options

The following ObjectCenter option affects the **cd** command:

path	Specifies the search path for loading source and object files (not for **#include** files) and for a matching pathname with the **cd** command.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

To facilitate loading and saving files, use the **cd** command to change the current working directory for ObjectCenter.

ObjectCenter searches the directories specified by the **path** option for subdirectories that match the pathname specified with the **cd** command.

Here is an example of the use of **cd** in connection with the **path** option:

```
-> pwd
 /my_home_directory
 -> printopt path
 path (unset)
 string - list of directories to search for source,
object, and library files
 -> cd temp3
 cd: cannot change to directory 'temp3'.
 -> setopt path ~/temp1/temp2
 -> cd temp3
 wd now: '/my_home_directory/temp1/temp2/temp3'
```

See Also **use**

CenterLine API

Application Program Interface to the CenterLine Engine

You can use the CenterLine API to integrate other tools with ObjectCenter.

The **CenterLine Engine** is an abstract component of an ObjectCenter environment that provides the following unique features:

- In-depth information about the internals of a program and its relationships, including data structures and functions
- Real-time execution of code fragments or complete applications with identification of run-time errors and immediate feedback on execution results

The **CenterLine API** is a programming interface to the CenterLine Engine. Figure 1 shows the relationships between the CenterLine API, the CenterLine Engine, and other abstract elements of ObjectCenter.

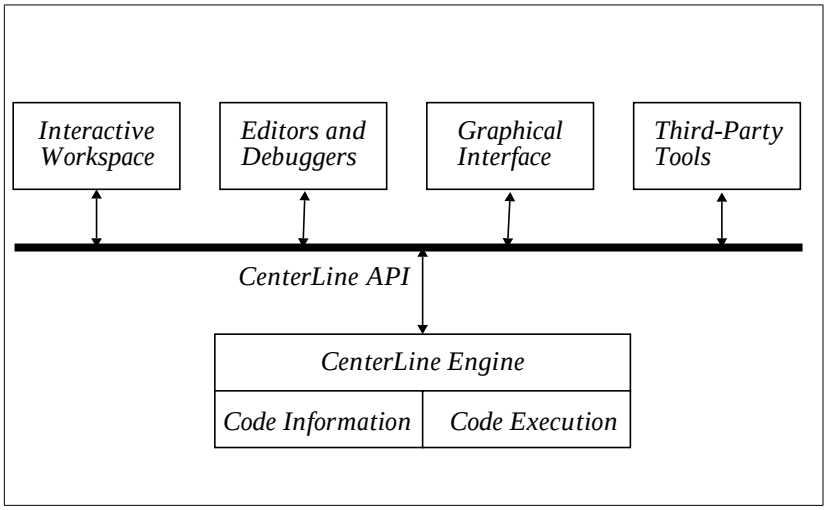

Figure 1 The CenterLine API and CenterLine Engine

The CenterLine API consists of a set of CenterLine Interprocess Communication (CLIPC) message definitions. CLIPC is the mechanism used by elements of ObjectCenter to exchange information with one another.

The **CenterLine/API/doc** directory contains detailed documentation describing CLIPC. The documents are provided in PostScript™ format, so you can view them with a PostScript previewer or print them on a PostScript printer.

See Also **CLIPC**

centerline_getopt()

returns the value of an option

cdm	pdm
✔	

Function syntax

extern "C" {char *centerline_getopt(char ****option*);}**

Usage

Use the **centerline_getopt()** function to return the value of *option* in a string. You can use this function to save the value of an option before changing it with the **setopt** command.

See Also

built-in functions, **printopt**, **setopt**, **unsetopt**

centerline_malloct()

allocates memory with type checking

cdm	pdm
✔	

Function syntax

```
extern "C" {
 void *centerline_malloct(unsigned int size);
}
```

Options

The following option affects the **centerline_malloct()** function:

save_memory Set this option if memory is scarce or for portions of a program that allocate very large arrays. If set, ObjectCenter does **not** use **the centerline_malloct()** function.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **centerline_malloct()** function to allocate memory on which run-time type checking is performed. If the **save_memory** option is not set, the standard C library functions **malloc()** and **calloc()** use **centerline_malloct()** to allocate memory, thereby silently providing type checking for all allocated memory.

The library functions **free(), cfree(),** and **realloc()** can be used with memory allocated by **centerline_malloct()**.

The **centerline_malloct()** function returns a pointer to a block of memory of *size* bytes. Whenever data is stored in this memory, ObjectCenter notes the type of the data. Later, when the memory is used, ObjectCenter checks that the type used to examine the data is consistent with the type used to store it.

centerline_[open | get | next | close]_sym

accesses symbol information

cdm	pdm
✔	

Function syntax

void *centerline_open_sym(char *_name_, int _code_);

Returns a handle to a set of symbol bindings that match *name*. The value of *code* must be **3**. If *code* is not **3**, the function returns **(void *) -1**.

void *centerline_get_sym(void *_cookie_, int _code_);

Returns a pointer to an appropriate piece of memory, depending on the following possible values of *code*

0	Returns a pointer to the character string corresponding to the name of the symbol.
1	Returns a pointer to the address of the data associated with the symbol. It will be 0 if the symbol is not defined.
2	Returns a pointer to a character string that contains the type declaration of the symbol.
3	Returns a pointer to a character string that contains the description of the storage class of the symbol; that is, whether it is **static** or **extern**.
other	Returns **(void*) -1**

The **centerline_get_sym()** function also returns **(void *) -1** if *cookie*, as returned from **centerline_open_sym()**, is not valid.

void *centerline_next_sym(void **cookie*);

Returns the following values:

-1	If *cookie* is invalid, or if there are no more symbols associated with *cookie*.
0	If there are more symbols, increments *cookie*'s pointer to the next symbol.

void *centerline_close_sym(void **cookie*);

Returns the following values:

-1	If *cookie* is invalid, or if there are no more symbols associated with *cookie*.
0	If *cookie* is valid; also frees up memory associated with *cookie*.

Example

Given the name of a symbol, the following code defines a function, **print_address()**, that prints the address of a symbol that ObjectCenter has loaded.

```
#include <stdio.h>

extern "C" {

int     centerline_open_sym();
int     centerline_get_sym();
int     centerline_close_sym();
void print_address(char * name);
}

void print_address(char * name)
    {
        int          cookie;
        int          address;

        cookie = centerline_open_sym(name, 3);
        if (cookie == 0)
        {
         puts("Error: cannot obtain symbol cookie.");
         return;
        }
```

```
    address = centerline_get_sym(cookie, 1);
    printf("Address of %s is 0x%08x\n",name,address);
    centerline_close_sym(cookie);
    return;
    }
```

Here is an example using the **print_address()** function from within ObjectCenter:

```
-> load print_address.c
Loading: print_address.c
-> print_address("hello");
Error: cannot obtain symbol cookie.
(void)
-> extern int i;
-> print_address("i");
Address of i is 0x00000000
(void)
-> int i;
-> print_address("i");
Address of i is 0x4013f478
(void)
-> &i;
(int *) 0x4013f478 /* i */
-> print_address("printf");
Address of printf is 0x403cf3a6
(void)
-> printf;
(int ()) 0x403cf3a6 < 'printf' module "/lib/libc.sl"
```

centerline_true()

indicates whether ObjectCenter is running

cdm	pdm
✔	

Function syntax

```
extern "C" {
int centerline_true(void);
}
```

Usage

Use the **centerline_true()** function to allow your programs to know at run time whether they are running in ObjectCenter. The function takes no arguments and returns the value 1.

To use it, create your own function called **centerline_true()** in its own source file. Your function must return 0.

```
int centerline_true(void) { return 0; }
```

Put that function in a library. When you compile and run the program from the shell, any calls to **centerline_true()** will use your version of the function and return 0. When you are working in ObjectCenter and your program calls **centerline_true()**, ObjectCenter's built-in function will be called instead of the function you wrote. This function returns 1. The built-in function is called even if you have attached the library containing your own version of **centerline_true()**.

Using this technique, you can test for the return value of **centerline_true()**. If it is 0, your program is not running in ObjectCenter; if it is 1, it is running in ObjectCenter.

centerline_unset()

marks memory as having unset value

cdm	pdm
✔	

Function syntax

```
extern "C" {
 int centerline_unset(void *addr, unsigned int  size);
}
```

Options

The following option affects the **centerline_unset()** function:

unset_value If set to **0**, tells ObjectCenter not to report variables used without being set.

See the **options** entry for more information about the **unset_value** option. ObjectCenter does not support this option in process debugging mode (pdm).

Usage

Use the **centerline_unset()** function to mark a region of *size* bytes starting at address *addr* as having unset value. Subsequent attempts to fetch values from the marked region yield warnings about unset values.

See Also

built-in functions, **centerline_untype()**, **X resources**

centerline_untype()

marks memory as initialized and valid

cdm	pdm
✔	

Function syntax

```
extern "C" {
 int centerline_untype(void *addr, unsigned int size);
}
```

Options

The following option affects the **centerline_untype()** function:

unset_value If set to **0**, tells ObjectCenter not to report variables used without being set.

See the **options** entry for more information about the **unset_value** option. ObjectCenter does not support this option in process debugging mode (pdm).

Usage

Use the **centerline_untype()** function to mark a region of *size* bytes starting at address *addr* as initialized, valid, and untyped. Subsequent attempts to fetch values from the marked region are accepted and do not yield unset value, type mismatch, or corrupted value warnings.

The **centerline_untype()** function works in much the same way as the **touch** command, but it is easier to call from a program and, unlike **touch**, will not mark invalid memory addresses.

Example

You can use the **centerline_untype()** function to deal with assignments in compiled code that are later the cause of inaccurate type mismatch warnings. Insert calls to **centerline_untype()** after the memory is referenced.

Here is an example:

```
char *mem_move(dest, src, size)
char *dest, *src;
int size;
 {
    char *orig_dest = dest;
    #if __CENTERLINE__
    int orig_size = size;
    #endif

    while (--size >= 0)
        *dest++ = *src++;
        #if __CENTERLINE__
        centerline_untype(orig_dest, orig_size);
        #endif
    return orig_dest;
 }
```

The value ObjectCenter uses to detect memory that has not been set is controlled by the **unset_value** option. The default value is **191**. You can change the value with the **setopt** command.

Restrictions

ObjectCenter initializes allocated data and local variables to the value **191** in order to perform checks on memory that is used before set. It is possible for spurious warnings to occur if the value **191** is stored in this memory while the program is executing within object code. Source code that uses **191** as a legitimate value will not generate spurious warnings.

Touching this memory will eliminate these spurious warnings. The warnings can also be suppressed with the **suppress** command, and the default value can be changed by modifying the **unset_value** option.

See Also

centerline_unset(), **X resources**

classinfo

displays information about classes

cdm	pdm
✔	

Command syntax

classinfo

classinfo *class_name*

Description

<< *none* >>	Motif and OPEN LOOK: Displays the Class Examiner. Ascii ObjectCenter: Sends information about all loaded classes to the Workspace.
class_name	Motif and OPEN LOOK: Displays the Class Examiner with the specified class. Ascii ObjectCenter: Sends information about the specified class to the Workspace.

Usage

Use **classinfo** to graphically display information about a class in the Class Examiner, when you are using the Motif or OPEN LOOK version of ObjectCenter. In Ascii ObjectCenter, **classinfo** sends class information to the Workspace.

Options

The following option affects the **classinfo** command:

show_inheritance Displays class members with the full inheritance path showing how members are inherited. If unset, shows a truncated inheritance path giving only the defining class.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

See Also **browse_base**, **browse_class**, **browse_data_members**, **browse_derived**, **browse_friends**, **browse_member_functions**, **list_classes**

clcc

invokes CenterLine's C compiler

The CenterLine-C compiler is a CenterLine product independent of ObjectCenter. If it is installed on your workstation, you can invoke the CenterLine-C compiler from within or outside of the ObjectCenter environment.

For usage information and a listing of the available compiler switches, issue the UNIX **man** command in the Workspace:

```
-> sh man clcc
```

You can also issue the **man** command at the shell:

```
$ man clcc
```

NOTE If you are using **clcc** in the ObjectCenter environment, be sure to read the "Specifying the search path for loading libraries and #include files" **TIP** on page 196.

See the *CenterLine-C Programmer's Guide* for additional information about the CenterLine-C compiler.

clezstart

shell command to invoke a utility that helps you load existing projects into ObjectCenter

Command syntax

clezstart *[switch ...] target ...*

where the current directory contains a makefile for *target*, and *switch* is a valid switch to be passed to the **make** command.

Description

The **clezstart** shell command invokes the EZSTART utility, which helps you load an existing project into ObjectCenter.

EZSTART uses existing makefiles to create another makefile, called **Makefile.cline**, which contains exactly the commands needed to load files into ObjectCenter for each target specified on the command line. The targets for **Makefile.cline** are typically named *target***_src** and *target***_obj**.

For instance, if you ordinarily use **make** to create a target named **my_project,** EZSTART helps you create ObjectCenter project components named **my_project_src** and **my_project_obj**.

Options

See Table 5 for a list of EZSTART options. Set these options by editing your **clezstart_init** file.

Table 5 EZSTART Options

Name of Option	What It Does
cl_ez_ar	If set to **a**, tells EZSTART to load files for a library individually. This allows you to swap individual library modules from object to source while debugging a library. See the “Making libraries” section on page 79 for more information.

Table 5 EZSTART Options (Continued)

Name of Option	What It Does
cl_ez_path	If set to **r**, tells EZSTART to generate relative pathnames for the **-I** and **-L** switches for compilers and linkers. By default, EZSTART generates absolute pathnames for these switches. Note that if you set this option to **r,** the **swap** command may not work correctly.
cl_ez_fstat	If set to **s**, suppresses checking on the existence of files to be loaded. By default, EZSTART checks to make sure that all of the files to be loaded really exist and issues a message in **Makefile.cline** if they do not exist; you could get this message if a file is moved or removed during the build process.
cl_nodebug_target	If set to **yes**, generates a third target named *target***_obj_nodebug**, which corresponds to the *target***_obj** target, except that the **nodebug** version is generated with the **-G** switch, excluding debugging information.

Usage

Use **clezstart** outside the ObjectCenter environment to create a new makefile for your project. Then you can use the **make** command in the ObjectCenter Workspace along with **Makefile.cline** to load your program as an ObjectCenter project.

When you install ObjectCenter, an **EZ** directory is created. The **clezstart_init** file in the **EZ** directory contains the environment variables, macros, and options that EZSTART requires. To modify this information for an entire system, edit **clezstart_init** in the **EZ** directory. To make local changes, copy the **clezstart_init** file into the directory where you plan to use **clezstart.** The **clezstart_init** file contains instructions on how to edit it.

By default, EZSTART recognizes the following tools: **cc**, **CC**, **gcc**, **acc**, **ld**, **make**, **ar**, **mv**, and **cp**.

Example

Here is an example of how to create an ObjectCenter project using a project already existing outside of ObjectCenter. See the “Scenarios” section on page 77 for more examples.

Begin by removing all the object files and executables that make up the targets you want to build. Many people put a **clean** target in the makefile for this purpose.

Then invoke **clezstart** from the directory where you normally execute **make**, giving the arguments to **clezstart** that you normally give to **make**.

For example, if you normally say:

```
$ make myProgram
```

enter:

```
$ clezstart myProgram
```

The **clezstart** script creates a **Makefile.cline** file in the directory from which you executed it. If a **Makefile.cline** file already exists, the following message appears:

```
File Makefile.cline already exists. Will not
overwrite. Rename or remove Makefile.cline.
```

If necessary, delete or rename the **Makefile.cline** file and re-execute **clezstart**.

Suppose your executable target in the makefile is called **myProgram**. There will be two targets in **Makefile.cline**: **myProgram_obj**, which loads all the object files and libraries that go into **myProgram**, and **myProgram_src**, which loads all the source files and libraries required by **myProgram**.

In the ObjectCenter Workspace, type the following:

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

and the appropriate object files will be loaded into ObjectCenter.

Finally, save your project by using the **save** command.

NOTE If, during the **make**, object files are moved to directories other than where they were compiled, the **swap** command will not work.

Also, you need to set the **swap_uses_path** and **path** options with the **use** command if the compilation occurs in a directory not containing the source file, as in the following example:

```
cc -c SubDir/x.c
```

Messages

After you run **clezstart**, you should examine the **Makefile.cline** file to see if it contains a message section with messages placed as comments. These messages indicate possible problems that may be encountered when using **Makefile.cline**. See Table 5 for a list of EZSTART messages and their meanings.

Table 6 EZSTART Messages

Text of Message	What the Message Means
# In target *target*: non-existent file -- *file*	A file (*file*) was used during the build that does not exist after the build has completed.
# In target *target*: Unable to determine source(s) for *file*	An object file was used in the build but no source file could be determined. You get this message if you do not do a complete **make clean** before you load the object files, or if the source for the object file is an assembler source (**.s** or **.S** suffix).
# The following command was captured, but no action taken: *#command*	EZSTART encountered a command that it cannot handle.

Table 6 EZSTART Messages (Continued)

Text of Message	What the Message Means
# In target *target* (_obj & _src): Load not done, unknown file type for *file* #<*command*>	EZSTART could not determine the type for *command*, which uses *file* as an input, and did not generate a load in either the **_obj** or the **_src** target. To correct this problem, review the command and insert the **load** commands manually if necessary.
# In target *target* (_obj & _src): No action taken on the following command: # *command*	Although EZSTART captured and identified *command* for the specified target, EZSTART could not determine what action to take.
# Encountered both **-Bstatic** and **-Bdynamic** linking	Both **-Bstatic** and **-Bdynamic** switches were specified in compilation or link commands. Check to make sure that they are properly used in **Makefile.cline**. See 'Using linker switches' on page 82 for more information.
# *file* was created multiple times -- check usage!	EZSTART created *file* more than once. It may be that a source file was compiled more than one time, with different settings, or **ar** was used many times to update a library. Look for occurrences of *file* in **Makefile.cline** to verify that they are correct.

Filename suffixes interpreted by clezstart

The **clezstart** utility interprets filename suffixes as follows:

.c	C or C++ source
.cc	C or C++ source
.C	C or C++ source
.cxx	C or C++ source
.i	C or C++ source

.o	Object file
.a	library
.so	library
.so.*	library
.s	Assembler source
.S	Assembler source
all others	Executable (or unknown)

Scenarios

In this section, we describe the following uses of **clezstart:**

- Building an executable target
- Using with non-standard tools
- Using absolute pathnames
- Building libraries
- Using different **make** systems
- Using recursive makes
- Adding commands

Building an executable target

If you are building an executable target and you wish to save time, instead of removing all the object files, just remove the executable file and run **clezstart**. Then the **Makefile.cline** file will contain two targets, but they will be the same in most cases; each target will load the object files and libraries that make up the executable at the highest level.

Using this method has one possible disadvantage: EZSTART does not capture the switches used for compiling source to object files.This may not be a problem for your project, especially if you use a consistent set of switches, because you can use **setopt** to set the **load_flags** option; see the **options** entry for more information about **load_flags.**

If you use additional tools

As previously mentioned, EZSTART recognizes the following by default: **cc**, **CC**, **gcc**, **acc**, **c89**, **clcc**, **ld**, **make**, **ar**, **mv**, and **cp**. If you use other tools, and if you do not invoke them by absolute pathname, you must use a few additional techniques to be successful with EZSTART.

Suppose you use a compiler other than **cc**, **CC**, **gcc**, or **acc**. You will need to create a link in the **EZ/tools** directory for that compiler. Alternatively, you can create the link in the directory from which you plan to invoke **clezstart.**

For example, if the C compiler you use is called **xcc**, do the following:

1. Use **cd** to change to the following directory:

   CenterLine/*arch_os***/EZ/tools**

   where *arch_os* designates the name of your particular platform.

2. Issue the following shell command:

   ```
   $ ln -s ./ezcc  xcc
   ```

3. Enter the following:

   ```
   $ clezstart
   ```

   to start up EZSTART as previously described.

When you use non-standard tools, link them according to the information in Table 7:

Table 7 EZSTART Links for Tools

Tool	Gets Linked to...	Example
C compiler	**ezcc**	`ln -s ezcc xcc`
C++ compiler	**ezCC**	`ln -s ezCC xCC`
linker	**ezld**	`ln -s ezld xld`
archiver	**ezar**	`ln -s ezar xar`

Using absolute pathnames for tools

If you invoke tools by specifying an absolute pathname, you must also do the following in order for EZSTART to work correctly:

- Use macros to represent the tool in your makefiles.
- Be consistent with the macro names. For instance, you cannot use **CC** for your C compiler in one makefile and **MYCC** in another makefile used during the same build.
- Use only one of each type of tool during the build. For instance, do not use **/bin/cc** and **/usr/local/bin/gcc** during the same build.

- Edit the **clezstart_init** file to indicate the correct names for the tools. For example, if you invoke your C compiler as **/usr/local/gnu/bin/gcc** and do this through the macro **CC**, edit **clezstart_init** in the following way:

Previous version of clezstart_init :	**Revised version of clezstart_init:**
cl_ezcc=	**cl_ezcc=/usr/local/gnu/bin/gcc**
cl_ezcc_macro=	**cl_ezcc_macro=CC**

- After you edit **clezstart_init**, you can execute **clezstart** as in the previous scenarios.
- Make sure that you edit lines in **clezstart_init** for all of the commands you invoke by absolute path.

Making libraries

Often libraries, or archives, are created as part of a build. The default behavior for EZSTART is to load the library when it is encountered by name in a command line—that is, as **myLib.a**, not **-lmyLib**.

If, however, you are working on debugging libraries, you probably want to have the individual component files of the library loaded instead of the library itself. This allows you to swap individual modules from object to source and back while you are debugging.

If you are sure that all archive commands which are executed during your build create libraries (not just add to them), you may want to edit **clezstart_init** to cause the component files from the libraries to be loaded into the ObjectCenter environment rather than having the library loaded. In order to do this, set the **cl_ez_ar** option to **a**:

```
cl_ez_ar=a
```

Using different make systems

If you use a program building system other than **make**, you may still be able to use EZSTART. You need to edit **clezstart_init**, changing the following line:

```
clezstart=
```

to

```
clezstart=<your make program>
```

You must also make a link in the **EZ/tools** directory to **ezmake**, naming the link with the name of your **make** program.

For example, if you use a **make** program called **xmake**, do the following from within the **EZ/tools** directory:

```
ln -s ./ezmake xmake
```

NOTE If you have recursive invocations of your **make** program, and do it by absolute path, you will not be able to use EZSTART.

Using EZSTART with recursive makes

If you have a project that requires many invocations of **make** on different makefiles, there are at least two ways you can use EZSTART.

In the first scenario, you can do a **make clean** for the whole system, then invoke **clezstart** at the top level. This will produce one **Makefile.cline** that will load all of the modules for the targets. This means that, if you build an object file in one directory from many smaller **.o** files in that directory (using the **-r** switch to **ld**), the individual **.o** files and corresponding source files will be loaded. This is the most complete and the most time consuming way of getting **Makefile.cline**.

A second scenario, if you are building libraries in many of the subdirectories, is to go to each of those and do a **make clean**, then invoke **clezstart** from that directory. This will create a **Makefile.cline** to load the individual components of the library. Next go to the top level and invoke **clezstart**; do not do a **make clean** in the directories where the libraries were created. Now the libraries will be loaded as archives, and when you are in ObjectCenter and you want to debug the files from the individual libraries, you can:

1. Unload the library
2. Use **cd** to change to the directory from which the library is made
3. Enter the following:

   -> **make -f Makefile.cline** *library target*

   to load the individual components into the environment.

Getting additional information placed in Makefile.cline

If you use commands such as **yacc** or **lex** in your builds, you may be able to capture the command line and have it placed in **Makefile.cline** as a comment in the messages area.

To do this, you must invoke the tool by name without using an absolute path—for instance, **yacc** and not **/bin/yacc**.

To cause the command line from the tool to be captured:

1. Use **cd** to change to the **EZ/tools** directory
2. Enter the following:

    ln -s ./cl_eztool *toolname*

For each tool or command linked in this way, a message will be placed in **Makefile.cline** every time it is invoked, along with the text of the command line. You can then edit **Makefile.cline** to supply additional information.

Restrictions

This section lists the known limitations of **clezstart.**

Changing your search path during a build

If the search path is changed during the build process, EZSTART may not work properly. This is because the scripts that are called reset the path to the value it had before **clezstart** was invoked and then call the real tool.

Some switches may not translate correctly

If you use a tool other than one of the defaults recognized by EZSTART that has different switches on its command line, EZSTART may not translate the command properly.

In most cases, EZSTART ignores switches that it does not know about. However, if the switch takes a value that is separated from the switch by spaces, EZSTART will probably interpret the value as an input file to the operation. As long as the value does not have the form of a valid input file, such as **foo.c** or **bar.o**, EZSTART will not generate a **load** command and will emit a message at the end of **Makefile.cline** informing you of the situation.

Some switches not recognized when using ar

It is possible that some switches will not be recognized correctly by EZSTART, depending on how you use **ar**, which is the UNIX command for maintaining groups of files into a single archive file. EZSTART tries to recognize when an archive is being updated or created. If you extract a file from an archive to be used in your build, the command will not be recognized.

Missing object files

If you produce an executable file by using a C or C++ compiler and include source files on the command line, EZSTART will attempt to load object (**.o**) files for the **_obj** target. Most compilers remove these files automatically in this case. For example, if you have the following generated by your **make**:

```
cc -o test test1.c test2.c -lm
```

Makefile.**cline** will have the following for the target **test_ob**j:

```
test_obj:
     #load test1.o
     #load test2.o
     #load -lm
     #setopt program_name test
```

Makefile.**cline** will also have two messages indicating that **test1**.**o** and **test2.o** do not exist, unless you have suppressed these messages with the **cl_ez_fstat** option.

In this case, when you try to make the **test_obj** target from within ObjectCenter, it fails. You can fix this problem by changing the way the executable is produced. Use the following two commands:

```
cc -c test1.c test2.c
ld -o test test1.o test2.o -lm
```

Requires Bourne shell

EZSTART requires **/bin/sh** to be the Bourne shell. If you have changed it to be some other shell, EZSTART will probably not work. If you have the Bourne shell in some other location, you can get EZSTART to work by editing all of the shell scripts in EZSTART and changing the first line of each to invoke the Bourne shell.

Using linker switches

If you use switches like **-B** more than once on the command line, ObjectCenter uses only one instance of the switch; this means that the results you get might not be what you expect.

For example, if you have the following:

```
cc -o prog prog.o -Bstatic -llib1 -Bdynamic -llib2
```

ObjectCenter generates the following load lines for the libraries:

```
#load -Bstatic -Bdynamic -llib1
#load -Bstatic -Bdynamic -llib2
```

You might need to edit the **Makefile.cline** file to correct this to the following, if it is what you intended:

```
#load -Bstatic -llib1
#load -Bdynamic -llib2
```

If you use both the **-Bstatic** and **-Bdynamic** options during the build, a message will be placed in the message file indicating that you should check the uses. There are cases where the results will not be correct. For instance, the same library name is used in two different links, one static and one dynamic.

Creating a file more than once

If you create a file more than once during the build (perhaps with different switch settings for a compilation), only the first one will be captured.

Using -n with clezstart

Specifying **-n** on the **clezstart** command line has no effect.

Updating a library more than once

If you update an archive more than one time during a build, the results will not be correct. In order to have EZSTART work properly in this case, do all updating of the archive with one command.

C library functions

C library functions replaced by ObjectCenter

To do its run-time error checking—such as checking for illegal indexes into arrays—and to make its environment behave like a standard UNIX process, ObjectCenter replaces many C library functions and system calls with its own version of them. For some of these functions you can substitute your own version. See your *ObjectCenter Platform Guide* for a list of the C library functions replaced by ObjectCenter.

Using your own function

To use your own version of a function, load the function in a source or object file before linking your program. If your program has already been linked, you must quit, then start a new ObjectCenter session to substitute your function for one of the ObjectCenter replacements.

NOTE To improve performance, you might want to substitute your own version for any library function; if you do so, however, you lose the error checking that ObjectCenter provides for that function.

Using libc versions of these functions

ObjectCenter loads its own version for each of these functions if the only other version is in a library ending with "**libc.a**". If you want to use the **libc** version, you must dearchive the function's module from **libc.a** and load it as a **.o** file, before linking or running your program in ObjectCenter.

CLIPC

CenterLine Interprocess Communication

CLIPC is the multi-cast message delivery service for exchanging data between elements of the ObjectCenter environment. ObjectCenter consists of multiple executables that use CLIPC messages to communicate with one another.

You can use CLIPC to integrate other tools with ObjectCenter. Using CLIPC is similar to using the Sun™ ToolTalk™ or HP SoftBench frameworks. For example, you can use CLIPC to integrate:

- Text editors other than **vi** and **emacs**
- A GUI builder
- A bug tracking and reporting system

Overview

CLIPC provides an abstract model for designing applications. An application consists of a set of application services, which can be dedicated to the application or shared among applications. Figure 2 shows the dedicated and shared services for ObjectCenter and an Email application. In the Figure, the Executive and Compiler Application Services constitute the CenterLine Engine, which provides information about program internals and execution.

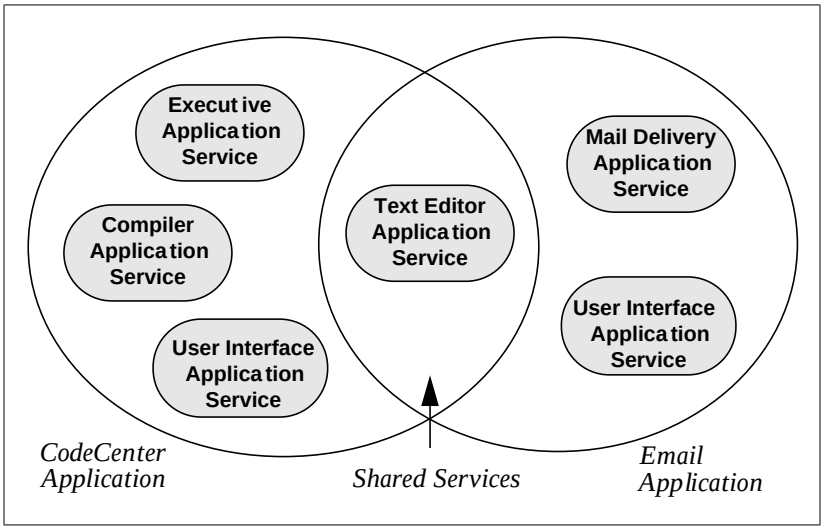

Figure 2 Dedicated and Shared Application Services

CLMS sessions

The CLIPC Message Server (CLMS), which is implemented as the **clms** process, manages the interprocess communication between all application services. A CLMS session consists of the **clms** process and a set of application service processes that are communicating on behalf of an application. The **clms** process and application service processes exchange CLIPC messages during the session. Both the **clms** and application processes are linked with the CLIPC interface library, a library of C functions for accessing elements of ObjectCenter.

Each CLMS session registers with the CenterLine registry service, a **clms_registry** process running on one or more server machines in the network. The registry keeps track of the location and members in each session in the network domain, so an individual application service can find its session. Along with the registry, CenterLine provides a shell command, **clms_query**, for displaying all the active sessions in a network. Figure 3 shows a sample session and the registry.

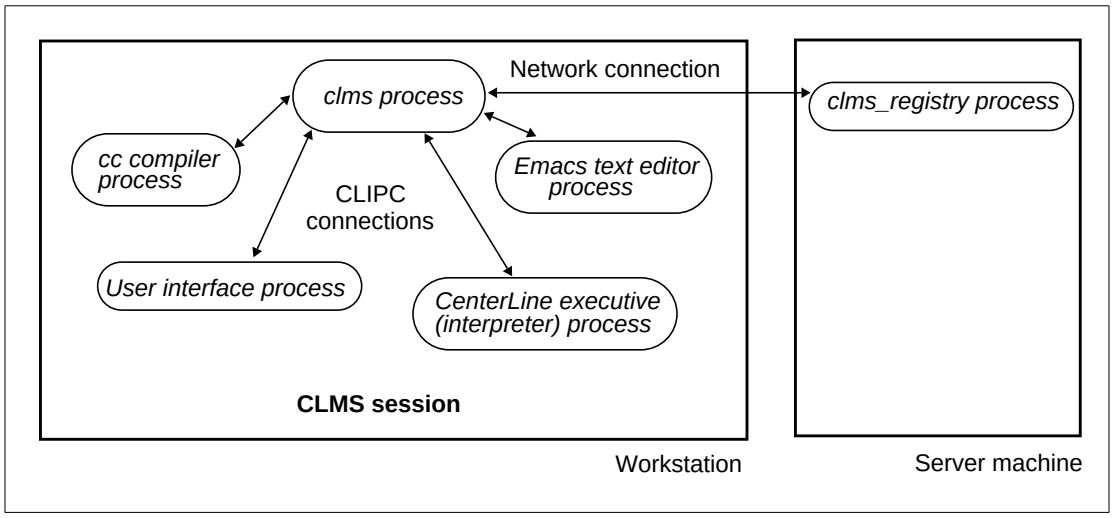

Figure 3 A Sample CLMS Session

Classes of application services

CLIPC application services fall into one or more of these classes:

- *Message producers* send CLIPC request and notification messages.
- *Message listeners* receive message types for which they have registered.
- *Request handlers* receive message types for which they have registered and send reply messages.

Although only one request handler can register for each message type, any number of application services can register as listeners.

Messages

A CLIPC message consists of a *name*, which identifies its type, an *envelope*, which provides delivery instructions, and a *body*, which contains the content of the message. CLIPC provides message types for all elements of ObjectCenter but, if desired, you can define new message types as well.

Message classes

CLIPC supports three message delivery classes in the envelope of the message:

- *Requests* initiate synchronous communication between a process that is requesting a service (a message producer) and a process that provides the service (a request handler).
- *Replies* confirm that the request handler performed the requested action for the message producer. Replies can include return values.
- *Notifications* indicate significant events, such as startup or termination of a process, and do not support verification of delivery. An application service can use a notification to send data to one or more application services. You can also use notifications for other purposes, such as implementing point-to-point links or a synchronous protocol.

Queueing and deadlock detection

CLIPC supports message queueing, but does not provide deadlock detection or avoidance. It is up to the application to provide such mechanisms when necessary.

Viewing CLIPC messages

CenterLine provides a human-readable message format called *message dump format* for reading CLIPC messages. The **clms_monitor** shell command displays all message traffic in a CLMS session in message dump format. This can be useful for debugging your application service as you integrate it with ObjectCenter.

Defining new message types

If you need to define new message types, use the CLIPC Message Definition Language, CMDL. CenterLine also provides a tool (**msg_parse**) for checking the syntax of message types coded in CMDL, and a tool (**msg_check**) for checking instances of messages in a running CLMS session against known message types.

Interface library

The CLIPC interface library provides a set of C language functions for exchanging CLIPC messages with the CenterLine Engine.

The library provides functions for

- Connecting to a CenterLine Engine session
- Registering to receive and handle messages
- Sending and receiving messages
- Allocating, getting, and assigning values to messages
- Memory management
- Diagnostics

Summary

To integrate your tool with ObjectCenter, you determine what information needs to be exchanged between the tool and ObjectCenter and design your application service. You determine which existing CLIPC messages to use and implement them in your code. You use the CLIPC functions in your code to participate in the CLMS session and exchange CLIPC messages. To debug your code, you use the **msg_check**, **clms_monitor**, and **clms_query** shell commands.

See Also

CenterLine API

The **CenterLine/API/doc** directory contains detailed documentation describing CLIPC. The documents are provided in PostScript format, so you can view them with a PostScript previewer or print them on a PostScript printer.

cmode

sets ObjectCenter to C mode

cdm	pdm
✔	

Command syntax **cmode**

Description

<< *none* >>	Sets ObjectCenter to C mode. Parses all statements you enter in the Workspace as C statements and treats them accordingly. Gives all output from commands in C syntax (not C++ syntax).

Usage

Use **cmode** to debug C source code and to display the internals of C++ data structures, such as virtual table pointers.

ObjectCenter is either in C++ mode or C mode at any time. C++ mode is the default. In C mode, all C++ identifiers are shown in mangled form, as they were generated by **cfront** when translating C++ code to C code.

NOTE The mode you are in affects only the way ObjectCenter accepts input and displays output in the Workspace. The mode does not affect how ObjectCenter loads source files. You can load C++ files even when in C mode.

NOTE See the **alias** entry on page 174 for more information about setting the default language for the environment using various options and commands.

Automatic mode switching

ObjectCenter automatically sets C mode when stopping execution in a C routine, going up or down the execution stack to a C routine, or resetting to a break level in a C module.

Prompt indicates mode

The Workspace prompt indicates the current mode, for example:

```
C++ 1 -> cmode
C Workspace Enabled.
C 2 ->
```

Working with C definitions and external definitions

If external functions and variables are not visible, you can switch to C mode to access them. C mode is also useful for being able to see structures internal to C++ (for example, to see detail about the inheritance path and to access information about the virtual function table).

With files loaded as C source code, no C definitions or external definitions are visible in the Workspace in C++ mode. Consider the C file **c_int.c**, which contains only this definition:

```
int c_var;
```

With C++ mode in effect, if you load **c_int.c** and ask for the value of the defined integer, ObjectCenter treats **c_var** as undefined:

```
C++ 55 -> load -C c_int.c
Loading (C): c_int.c
C++ 56 -> c_var;
Error #719: c_var undefined.
```

The workaround is to use C mode:

```
C++ 57 -> cmode
C Workspace Enabled.
C 58 -> c_var;
(int) 0
```

See Also

cxxmode, **language selection**

NOTE ObjectCenter no longer supports the **-p** switch. By default, all file descriptors that are open when you start ObjectCenter remain open during your session.

code generation

generating K&R C or ANSI C intermediate code in ObjectCenter

ObjectCenter is based on the USL C++ Language System. The USL C++ Language System uses cfront to translate C++ source code into C intermediate code before compilation.

Because ObjectCenter is based on the USL C++ Language System, ObjectCenter generates C intermediate code internally in order to interpret C++ source code in the ObjectCenter environment. The type of C intermediate code ObjectCenter generates is either K&R or ANSI C. K&R C intermediate code generation is the ObjectCenter default. You can choose ANSI C intermediate code generation at startup or toggle between the two types in a session, as described later in this entry.

Generating C intermediate code does not apply to

- Selecting an underlying compiler for ObjectCenter

  You can change the underlying compiler from the default compiler (CC) to any one of a number of K&R or ANSI C compilers. To do this, use the **config_c_parser** command (see the **config_c_parser** entry) and change the **config_c_parser** option to C (see the **options** entry).

  The **config_c_parser** command and the **primary_language** option apply only to changing the underlying compiler.

- Creating files outside the ObjectCenter environment

  ObjectCenter generates intermedicate C code internally in order to interpret C++ source code in the ObjectCenter environment. ObjectCenter produces no files for use outside ObjectCenter. To produce object files and executables for use outside ObjectCenter, you must use an external tool, such as CC. CC produces either K&R or ANSI C intermediate code depending on whether you use the **+a1** switch (see the **CC** entry on page 44).

Usually, generating K&R C intermediate code (the default) causes no problems for programs loaded in the ObjectCenter environment. Calling functions from the Workspace, however, may cause incorrect results if the functions are in object form produced by a compiler that generated ANSI C intermediate code, and the functions pass **float**s, **short**s, or **char**s by value.

Run-time errors may also occur if both the following are true.

- You link C++ source code with object code produced by a compiler that generated ANSI C intermediate code; or you link object code produced by a compiler generating K&R C intermediate code with object code produced by a compiler generating ANSI C intermediate code.

  and

- Your program passes **float**s, **short**s, or **char**s by value as function arguments. K&R and ANSI C handle the promotion of integers in different ways. See Section 4.1 of the *Annotated C++ Reference Manual* by Ellis and Stroustrup.

If ObjectCenter produces these incorrect results because ObjectCenter generates K&R C intermediate code by default, you can select ANSI C intermediate code generation.

For example, suppose ObjectCenter produced incorrect results because you loaded source code into ObjectCenter with object code produced by the Sun C++ Compiler 3.0.1 (bundled with SparcWorks 2.0.1). Sun C++ Compiler 3.0.1 produces ANSI C intermediate code. Instead of recompiling the object code and reloading it into ObjectCenter, you can select ANSI C intermediate code generation in ObjectCenter.

Selecting ANSI C intermediate code generation, however, may still not produce correct results if your program has a mix of object files and they pass **float**s, **short**s, or **char**s by value as function arguments.

To select ANSI C intermediate code generation in ObjectCenter

You can select ANSI C intermediate code generation in ObjectCenter in one of two ways: by using the **-backend_ansi** command-line switch when you invoke ObjectCenter or by setting the **backend_ansi** option during a session.

At startup

To select ANSI C intermediate code generation at startup, use the **-backend_ansi** command-line switch when you invoke ObjectCenter or set the **backend_ansi** option in your **.ocenterinit** (startup) file.

To use the **-backend_ansi** command-line switch, enter:

```
% objectcenter [other switches] -backend_ansi
```

Using the **-backend_ansi** switch at startup automatically loads the ANSI C **libC** (the standard C++ library) and sets the **backend_ansi** option.

To set the **backend_ansi** option in your **.ocenterinit file**, add this line to it:

```
setopt backend_ansi
```

Using the **backend_ansi** option in your **.ocenterinit** file automatically loads the ANSI C **libC**.

During a session

To toggle to ANSI C intermediate code generation during a session, set the **backend_ansi** option.

1. Unload the K&R C version of **libC**.

   ```
   C++ -> unload -lC
   ```

2. Enter this in the Workspace:

   ```
   C++ -> setopt backend_ansi
   ```

3. Load the ANSI C version of **libC**.

```
C++ -> load -L path-to/CenterLine/clc++/archos/lib/a1 -lC
```

To toggle back to K&R C intermediate code generation during a session, unset the **backend_ansi** option.

1. Unload the ANSI C version of **libC**.

   ```
   C++ -> unload -lC
   ```

2. Enter this in the Workspace:

   ```
   C++ -> unsetopt backend_ansi
   ```

3. Load the K&R C version of **libC**.

```
C++ -> load -L<path-to>/CenterLine/clc++/<archos>/lib/a0 -lC
```

Changing the **backend_ansi** option during a session affects only the files you load after you change the option. If you want the option to affect a file already loaded, you must unload the file and load it after you have made the change to the option.

Use the -backend_ansi switch in most cases

Using the **-backend_ansi** command-line switch is preferable in most situations, because the switch automatically loads the correct library at startup, and ObjectCenter generates ANSI C intermediate code for all the source files you load during a session.

To use the **backend_ansi** option during a session, you must unload and load libraries and unload and load files for which you want ObjectCenter to generate the other type of intermediate C code. You may want to use the **backend_ansi** option only in those situations where you do not want to restart the session, and you know which source files are causing problems.

Effects of the backend_ansi option

Setting and unsetting the **backend_ansi** option has the following effects::

	backend_ansi set	**backend_ansi** unset
When loading **libC**	Load ANSI C **libC**	Load K&R C **libC**
When invoking **CC**	Use **+a1**	Use **+a0**
Setting of **ansi** option (affects preprocessing)	Set (except that __STDC__ remains undefined)	Unset
When loading C++ source	Don't promote **float** arguments to **double**	Promote **float** arguments to **double**
	Don't promote unsigned **short**s and unsigned **char**s to unsigned **int**s	Promote unsigned **short**s and unsigned **char**s to unsigned **int**s

Examples

At startup, invoking ObjectCenter without the **-backend_ansi** switch loads K&R C **libC** and generates K&R C intermediate code:

```
% objectcenter
Attaching
<path-to>/CenterLine/clc++/<archos>/lib/a0/libC
C++ -> load /my_dir/K_R/*.o
C++ -> load /my_dir/src/*.C
C++ -> link
```

At startup, invoking ObjectCenter with the **-backend_ansi** switch loads ANSI C **libC** and generates ANSI C intermediate code.

```
% objectcenter -backend_ansi
Attaching
<path-to>/CenterLine/clc++/<archos>/lib/a1/libC
C++ -> load /my_dir/ANSI/*.o
C++ -> load /my_dir/src/*.C
C++ -> link
```

During a session, toggling from ANSI C intermediate code generation back to K&R C might look like this:

```
% objectcenter -backend_ansi
Attaching
<path-to>/CenterLine/clc++/<archos>/lib/a1/libC
C++ -> load /my_dir/ANSI/*.o
C++ -> load /my_dir/src/*.C
C++ -> link
C++ -> run
C++ -> unload user
C++ -> unload -lC
/* unload the ANSI C libC currently loaded */
Unloading
<path-to>/CenterLine/clc++/<archos>/lib/a1/libC
/* then load the K&R C libC */
C++ -> unsetopt backend_ansi
C++ -> load
-L<path-to>/CenterLine/clc++/<archos>/lib/a0 -lC
 Attaching
<path-to>/CenterLine/clc++/<archos>/lib/a0/libC
C++ -> load /my_dir/K_R/*.o
C++ -> load /my_dir/src/*.C
```

See Also

CC, **load**, **objectcenter**, **options**

commands

ObjectCenter provides a complete set of commands that you can issue from the Workspace in either component debugging mode or process debugging mode.

Component and process debugging modes

In *component debugging mode*, you load all the parts, or components, of your program and link and execute them within ObjectCenter. In contrast, in *process debugging mode*, you load your program as a fully linked executable, and you have the choice of debugging it along with a corefile or attaching to another process.

Issuing commands from the Workspace

Issue commands from the Workspace by typing them at the Workspace prompt. When you are in ObjectCenter's default mode, which is component debugging mode, the prompt is a right arrow:

```
->
```

If you are in process debugging mode, ObjectCenter lets you know by adding **pdm** to the right-arrow prompt:

```
pdm ->
```

See the **pdm** entry on page 254 for more details about debugging in process debugging mode.

Most ObjectCenter commands are also available in either C++ mode (the default) or in C mode. To let you know which mode you are in at any point, ObjectCenter has a distinct Workspace prompt for each mode:

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

Issuing commands from Motif or OPEN LOOK

Besides using the Workspace to issue commands, you can issue most commands by using ObjectCenter's Graphical User Interface; see Table 8 on page 97 for information about where you can find each command in a menu. Consult the *User's Guide* for details about using the Motif or OPEN LOOK versions of ObjectCenter commands.

Using commands as functions in your code

ObjectCenter provides predefined function equivalents of all commands; you can use these functions in C++ or C programs. See the **built-in functions** entry on page 34 for more information.

Defining your own commands

You can define your own ObjectCenter commands for use in the Graphical User Interface; see the **X resources** entry on page 436 for more information.

List of commands and menus

Table 8 lists the ObjectCenter commands, a brief description of each, and the mode in which they can be used (cdm, pdm, or both).

NOTE See the entry for each command for more details about that particular command.

Table 8 Brief Description of ObjectCenter Commands[a]

Command Name	cdm	pdm	Brief Description
action	✔		Sets a debugging action
alias	✔	✔	Creates an alias for a command
assign	✔	✔	Assigns a value to a variable
attach		✔	Attaches to a running process
browse_base	✔		Displays base classes in the Workspace
browse_class	✔		Displays base and derived classes and members in the Workspace
browse_data_members	✔		Displays data members in a class in the Workspace
browse_derived	✔		Displays derived classes in the Workspace
browse_friends	✔		Displays friend classes and friend functions in the Workspace
browse_member_functions	✔		Displays member functions in a class in the Workspace
build	✔	✔	Reloads all files in the project that have changed
catch	✔	✔	Traps signals before they reach the program
cd	✔	✔	Changes the current working directory

Table 8 Brief Description of ObjectCenter Commands[a] (Continued)

Command Name	cdm	pdm	Brief Description
classinfo	✔		Displays information abut classes
config_c_parser	✔		Specifies the compiler configuration to use for load-time error checking
construct	✔		Invokes constructors for static objects
cont	✔	✔	Continues execution from a break level
contents	✔	✔	Lists files in the current project
cxxmode	✔		Sets ObjectCenter to C++ mode
debug		✔	Loads an executable file, a corefile, or a process for debugging
delete	✔	✔	Deletes debugging items
destruct	✔		Invokes destructors for static objects
detach		✔	Detaches from a running process
display	✔	✔	Displays the value of a variable or expression
down	✔	✔	Moves down the execution stack
dump	✔	✔	Displays all local variables
edit	✔	✔	Invokes your editor at a specified location
email	✔	✔	Sends electronic mail to CenterLine Software
expand	✔		Lists the functions that could be called by a C++ statement
english	✔		Describes a C type in English (not available in C++ mode)
fg	✔		Returns to ObjectCenter after **suspend**
file	✔	✔	Displays and sets the current list location
gdb		✔	Executes a **gdb** command

Table 8 Brief Description of ObjectCenter Commands[a] (Continued)

Command Name	cdm	pdm	Brief Description
gdb_mode		✔	Changes from **pdm** mode to **gdb** mode
help	✔	✔	Displays usage information about commands
history	✔	✔	Lists previously entered input
ignore	✔	✔	Allows signals to pass directly to the program
info	✔		Displays information (address, name, size, and type) for data at a specific memory location
instrument	✔		Enables run-time error checking for an object file
keybind	✔		Changes bindings used by the in-line editor in the Workspace
link	✔		Links files from libraries
list	✔	✔	Displays source code lines
list_classes	✔		Lists the names of all classes that are loaded
listi		✔	Displays machine instructions
load	✔		Loads source, object, library, and project files
load_header	✔		Loads header files as source
make	✔	✔	Invokes the UNIX **make** command to handle CenterLine (CL) targets
man	✔	✔	Displays information about ObjectCenter and UNIX items
next	✔	✔	Executes source code by line; does not enter functions
nexti		✔	Executes machine code by line; does not enter functions
print	✔	✔	Prints the value of variables and expressions
printenv	✔	✔	Displays the system environment

Table 8 Brief Description of ObjectCenter Commands[a] (Continued)

Command Name	cdm	pdm	Brief Description
printopt	✔	✔	Displays information on ObjectCenter options
proto	✔		Generates prototypes for C functions and writes them to a file
quit	✔	✔	Quits ObjectCenter
reinit	✔		Initializes all global variables
rename	✔		Rename an ObjectCenter function
rerun	✔	✔	Executes **main()** with new arguments
reset	✔	✔	Returns to a previous break level (**cdm**) or to the top level (**pdm**)
run	✔	✔	Executes **main()** with arguments
save	✔		Saves the current session in a project file
set	✔	✔	Assigns a value to a variable
setenv	✔	✔	Adds a variable to the system environment
setopt	✔	✔	Sets an ObjectCenter option
sh	✔	✔	Executes a Bourne subshell
shell	✔	✔	Executes a subshell
source	✔	✔	Reads ObjectCenter commands from a file
start	✔		Executes **main()** without initializing global variables
status	✔	✔	Lists debugging items (actions, breakpoints, displayed items, and traces)
step	✔	✔	Steps execution by statement, entering functions
stepi		✔	Steps execution in machine instructions by statement, entering functions

Table 8 Brief Description of ObjectCenter Commands[a] (Continued)

Command Name	cdm	pdm	Brief Description
stepout	✔	✔	Continues execution until the current function returns
stop	✔	✔	Sets a breakpoint
stopi		✔	Sets a breakpoint at a machine instruction
suppress	✔		Suppresses reporting of a warning
suspend	✔		Suspends ObjectCenter and returns to the shell
swap	✔		Replaces a file or function with its source / object counterpart
thread		✔	Sets a thread to be the current one or affects the display of information about a thread
threads		✔	Displays information about threads active at a break location
touch	✔		Marks memory as initialized and valid
trace	✔		Traces program execution
unalias	✔	✔	Removes an alias for a command
uninstrument	✔		Disables run-time error checking for an object file
unload	✔		Unloads files
unres	✔		Lists undefined variables and functions
unsetenv	✔	✔	Removes a variable from the system environment
unsetopt	✔	✔	Unsets an ObjectCenter option
unsuppress	✔		Reactivates reporting of a warning
up	✔	✔	Moves up the execution stack
use	✔	✔	Displays or sets the directory search path
whatis	✔	✔	Lists all uses of a name

Table 8 Brief Description of ObjectCenter Commands[a] (Continued)

Command Name	cdm	pdm	Brief Description
when		✔	Executes specified commands under specified conditions
where	✔	✔	Displays the execution stack
whereami	✔	✔	Displays the current break and scope locations
whereis	✔	✔	Lists the locations where a name is declared or defined
xref	✔		Cross-references a function or variable

a. The following commands are no longer supported by ObjectCenter: **jobs**, **kill**, and **userprint**.

config_c_parser

specifies the compiler configuration to use for load-time error checking and code generation

cdm	pdm
✔	

Command syntax

config_c_parser

config_c_parser *config*

Description

<< none >> Displays the C compiler configuration currently being used for load-time error checking.

config Specifies the C compiler configuration to use for load-time error checking.

Here are some possible values for *config*:

suncc Sun's K&R C compiler

acc Sun's ANSI C compiler

hpcc Hewlett-Packard's K&R C compiler

hpc89 Hewlett-Packard's ANSI C compiler

gcc The GNU C compiler provided by the Free Software Foundation

clcc CenterLine-C compiler

NOTE See the **language selection** entry on page 175 for more information about setting the default language for the environment using various options and commands.

The **config_c_parser** command has no effect on header files or libraries used by ObjectCenter; see the "Specifying the search path for loading libraries and #include files" **TIP** on page 196 for more information about these specifications.

Usage

Most non-ANSI C compilers have differing parsing and code generation rules for certain C constructs. Moreover, many ANSI compilers differ in areas where the standard is ambiguous or implementation-dependent. ObjectCenter's goal is to provide error messages and generate code that is consistent with your C compiler. Use the **config_c_parser** command to specify the default C compiler configuration that you want ObjectCenter to emulate when you load your source file.

For instance, if you plan to compile your source code with the CenterLine-C compiler, you would typically set the compiler configuration to **clcc**. Then, ObjectCenter would use the same rules as the CenterLine-C compiler in generating load-time errors.

If an ObjectCenter option such as **ansi** is set, the option takes precedence over the default configuration. Suppose, for instance, you issue the **config_c_parser** command with *config* as **clcc** and set the **ansi** option. The resulting configuration enables ANSI mode, even though by default, the **clcc** configuration disables ANSI mode. See the following example:

```
-> config_c_parser clcc
-> config_c_parser
The current base parser configuration is: clcc
NO : ANSI mode
...
-> setopt ansi
-> config_c_parser
The current base parser configuration is: clcc
YES : ANSI mode
...
```

NOTE The **ansi** and the **long_not_int** options are the only two ObjectCenter options that affect the compiler configurations displayed or set by the **config_c_parser** command.

See Table 9 for a listing of the default configurations.

Table 9 Default C Compiler Configurations Supported by ObjectCenter

Configuration Items	**Name of C Compiler**					
	suncc	**acc**	**hpcc**	**hpc89**	**gcc**	**clcc**
ANSI mode	NO	YES	NO	YES	NO	NO
Initialization of automatic aggregates	NO	YES	NO	YES	YES	YES
Bitfields are unsigned by default	YES	YES	YES	YES	NO	YES
Type **char** is unsigned by default	NO	NO	NO	NO	NO	NO
Type **long** is equivalent to type **int**	YES	NO	NO	NO	YES	NO
Follow ANSI rules for promoting arithmetic operands	NO	NO	NO	YES	YES	YES
Follow ANSI rules for promoting function arguments	NO	YES	NO	YES	YES	YES
size_t is type **unsigned int**	NO	NO	YES	YES	NO	NO
size_t is type **signed int**	YES	YES	NO	NO	YES	YES
size_t is type **unsigned long**	NO	NO	NO	NO	NO	NO
size_t is type **signed long**	NO	NO	NO	NO	NO	NO

See Also **language selection**

construct

invokes constructors for static objects

cdm	pdm
✔	

Command syntax

construct

construct *file*

Description

<< *none* >>	Invokes static constructors for all loaded files.
file	Invokes static constructors for the specified file.

Usage

Use **construct** to invoke constructors for static objects. The **construct** command does not affect objects created in the Workspace.

TIP: How are construct, reinit, and start commands related to run and rerun?

When you issue the **run** command for a C program in ObjectCenter, you are implicitly calling two commands: the **reinit** and the **start** commands. The **reinit** command causes global variables and signal handlers to be initialized, frees allocated memory, and closes open files. The **start** command causes **main()** to execute without automatically initializing any variables or closing any files.

For a C++ program, the initialization process is more complicated because of the use of constructors to initialize class objects. When you issue a **run** command for a C++ program, you implicitly call three commands rather than two; they are **reinit, construct,** and **start**. The **construct** command invokes the constructors for static objects.

See Also

destruct

cont

continues execution from a break level

cdm	pdm
✔	✔

Command syntax

cont

cont *continuation_value*

cont at *line*

cont at *line* **sig** *signum*

cont sig *signum*

cont skip *count*

Description

<< *none* >>	Continues execution of the program from the current break level.
continuation_value	When a break level was created because of a warning or error, substitutes *continuation_value* for the expression that caused the error or warning and continues execution of the program. Using a continuation value as an argument allows execution to continue beyond an expression that generates an error or warning. (**cdm** only)
at *line*	Continues at location specified by *line. (***pdm** only)
at *line* **sig** *signum*	Continues at location specified by *line* with signal specified by *signum*. This means the signal is delivered to your program, which must handle it. *(***pdm** only)
sig *signum*	Continues with signal specified by *signum. (***pdm** only)
skip *count*	Continues, ignoring breakpoint for *count* iterations. *(***pdm** only)

Usage

Use the **cont** command to continue execution of the program from a break level.

Control-d is a keyboard shortcut for calling **cont** without an argument.

Restrictions

You cannot continue from all errors by supplying a continuation value to **cont**.

See Also

step, **stepout**, **stop**, **where**, **whereami**

contents

lists files in the current project or source file

cdm	pdm
✔	✔

Command syntax

contents

contents -ascii

contents all

contents *file*

Description

<< *none* >>	Returns the pathname of the **a**.**out** file currently loaded. (**pdm** only)
	Ascii ObjectCenter: Lists all files that you have attempted to load—both those that loaded successfully and those that failed to load. (**cdm** only)
	Motif and OPEN LOOK: Invokes the Project Browser.
all	In addition to all the files that you have attempted to load, lists library modules that have been linked during the session. (Ascii ObjectCenter only)
	Lists known source files for the **a.out** file currently being debugged. (**pdm** only)
file	If the name of a loaded file, lists all objects declared or defined in the file.
	If the name of a static library, lists all modules that have been linked from the library.
	If the name of a shared library, lists all symbols in the library.
	Lists the functions defined in the source file named *file.* (**pdm** only)

Switches

-ascii Motif and OPEN LOOK: Displays the output of the **contents** command in ASCII format in the Workspace. Without this switch, the **contents** command invokes the Project Browser in Motif and OPEN LOOK. (**cdm** only)

Usage

Use the **contents** command to display information about files in your current project. The **contents** command lists only the files that were compiled with debugging information (with the **-g** switch).

Restrictions

The **contents** command does not list project files.

The following restrictions apply in process debugging mode:

- The **contents all** variation does not display files compiled without debugging information.
- The **contents** *file* variation may return only a partial list of objects declared or defined in *file*.

See Also

build, **load**, **make**, **swap**, **unload**

cxxmode

sets ObjectCenter to C++ mode

cdm	pdm
✔	

Command syntax

cxxmode

Description

<< none >> Sets ObjectCenter to C++ mode. Parses all statements you enter in the Workspace as C++ statements and treats them accordingly. Gives all output from commands in C++ syntax (not C syntax) using C++ names.

Usage

Use **cxxmode** to debug C++ code.

ObjectCenter is either in C++ mode or C mode at any point. C++ mode is the default. In C++ mode, identifiers are shown in C++ form, not in mangled form as they were generated by **cfront** when translating C++ code to C code.

NOTE The mode you are in affects only the way ObjectCenter accepts input and displays output in the Workspace; it does not affect how ObjectCenter loads source files. You can load C files even when in C++ mode. See the **alias** entry on page 8 for more information about setting the default language for the environment using various options and commands.

Automatic mode switching

ObjectCenter automatically sets C++ mode when stopping execution in a C++ routine, going up or down the execution stack to a C++ routine, or resetting to a break level in a C++ module.

Prompt indicates mode

The Workspace prompt indicates the current mode, for example:

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

Working with C definitions and external definitions

With files loaded as C source code, no C definitions or external definitions are visible in the Workspace in C++ mode.

Consider the C file **c_int.c**, which contains only this definition:

```
int c_var;
```

With C++ mode in effect, if you load **c_int.c** and ask for the value of the defined integer, ObjectCenter treats **c_var** as undefined:

```
C++ 55 -> load -C c_int.c
Loading (C): c_int.c
C++ 56 -> c_var;
Error #719: c_var undefined.
```

The workaround is to use C mode:

```
C++ 57 -> cmode
C Workspace Enabled.
C 58 -> c_var;
(int) 0
```

See Also

cmode, **language selection**

debug

loads an executable file, a corefile, or a process for debugging

cdm	pdm
	✔

Command syntax

debug

debug *executable*

debug *executable corefile*

debug *executable process_id*

Description

<< *none* >>	Displays the name and arguments of the program being debugged.
executable	Loads the symbol table for *executable*, which is the name of the executable program to be debugged.
executable corefile	Loads the symbol table from the executable program (*executable*) and sets up ObjectCenter to work with the *corefile* along with the executable program. The *corefile* contains a literal copy of the contents of memory at the time that the operating system aborted a program.
executable process_id	Loads the symbol table from the *executable* file and attaches to the running process identified by *process_id*. The process can be running inside or outside of ObjectCenter.

Options

class_as_struct	Disables maximum processing of classes to improve performance
full_symbols	Forces the reading of the full symbol table for maximum information immediately.

These options are only available in process debugging mode.

Usage

Use the **debug** command (in process debugging mode) to load the files required for the following kinds of source-level and machine-level debugging:

- Source-level debugging of a fully linked executable program
- Machine-level debugging a fully linked executable program along with a corefile
- Source-level debugging a running process

These debugging activities are not available with ObjectCenter unless you are in process debugging mode.

Using the -g switch

The information in the symbol table in an executable file varies according to whether or not you used the **-g** switch when you compiled the object modules that you linked to create it. Modules that are not compiled with **-g** contain the information for machine-level debugging only, plus information about the hexadecimal address of external symbols. Modules compiled with **-g**, in contrast, contain full source-level debugging information. Also, if you strip debugging information from an executable file, you are limited to machine-level debugging without any knowledge of external symbols.

Using run after debug

After you use **debug** to load a program, use **run** to start it running. This causes ObjectCenter to create a process and make that process run your program. You can then use any ObjectCenter commands that are available in process debugging mode to debug the program.

If your program crashes and creates a corefile, you can use **debug** to load the corefile created when it crashed.

Attaching to a process

When you attach to a running process, the first thing that ObjectCenter does is to stop the process. You can then examine and modify the process with the commands available in ObjectCenter. If you want the process to continue running, use the **continue** (**cont**) command. Use the **detach** command to release a process from ObjectCenter's control.

You can use the **attach** command in combination with **debug** to attach to an already running process. That is, you can use the following two commands:

```
(pdm)  1 -> debug my_executable
(pdm)  2 -> attach my_process_id
```

instead of the following:

```
(pdm) 3 -> debug my_executable my_process_id
```

NOTE If you leave process debugging mode or use the **run** command while you have an attached process, you kill that process.

See Also **attach**, **detach**, **pdm**

debugging

balancing speed and other trade-offs with various ObjectCenter debugging capabilities

ObjectCenter allows you to perform the following kinds of debugging activities:

- Load-time error checking of source code
- Run-time error checking of source and object code
- Interactive source-level debugging of source, object, library, and **a.out** files
- Code visualization through examination of class hierarchies, individual classes, cross-references and data variables

Different kinds of debugging help you find different kinds of errors. Also, there are trade-offs with each kind of debugging; for instance, you get the most thorough error checking when you load source code, but source code is the slowest to execute in ObjectCenter.

We assume that you will use the various kinds of debugging at various stages of the development process. In the rest of this reference page, we describe scenarios that illustrate different ways to balance speed of execution with run-time error checking and debugging capabilities. We also provide a brief overview of all the debugging activities listed above, along with performance considerations associated with each activity.

Loading source versus object code versus executables

Table 10 summarizes the pros and cons for each of six scenarios. Each scenario balances speed with run-time error-checking and debugging in a different way. We discuss each scenario in more detail after the table. For the best performance in most debugging scenarios, we recommend that you compile object files with the **-dd=on** compile switch. For browsing and developing new code, on the other hand, you might want to compile with **-dd=off**. See the **demand-driven code generation** entry on page 129 for more information about this feature.

Table 10 Six Debugging Scenarios Showing Trade-Offs for Loading Source vs. Object Code

	Files Used in this Scenario	Debugging Considerations with this Scenario		
		Run-Time Error Checking	**Speed of Execution**	**Debugging Available**
1	Load all modules as source files	excellent	slowest	excellent
2	Load 1 or 2 as source files; load the rest as instrumented object code[a]	very good	somewhat less than full speed	excellent
3	Load all files as instrumented object code	good	somewhat less than full speed	good
4	Load all files as regular object code with debugging information (compiled with **-g** switch)	minimal	full speed	good
5	Load all files as regular object code without debugging information (compiled without **-g**, or loaded with **-G**)	none	full speed	minimal
6	Load the **a.out** file in **pdm**	none	full speed	good

a. See the "Run-time error checking" section on page 121 for a definition of instrumented object code.

Scenario #1: Maximizing number of errors reported

Suppose your goal is to catch as many run-time errors as possible, no matter what the cost in processing speed. In this case, you could load your program entirely as source code, correcting or suppressing load-time warnings, then build and run your program.

This scenario costs the most execution time, but it yields the most complete set of run-time errors. This scenario is really useful only with small projects. Some projects may be so large that this scenario is not possible.

#2: Getting fewer run-time errors but increasing speed

The second scenario is much more typical for a large C++ program; we recommend either this scenario or the third scenario for most applications.

Let us assume that you cannot afford the execution time to load and run your program entirely as source code, but you suspect that one or two source modules are likely to have errors. In this scenario, you load the suspect modules as source and the other modules as instrumented object code. Then, when you link and run your program, you will probably catch many, if not all, run-time errors, and your program's execution time will not be nearly as great as in the first scenario.

#3: Increasing speed even more

To improve speed even more than the second scenario, load your program entirely as instrumented object code. Your program will run only slightly slower than the full speed of the machine, and you will get much, though not all, of the run-time error checking and debugging capabilities ObjectCenter provides. This scenario may be the most typical and effective way to use ObjectCenter for many users.

#4 and #5: Maximizing speed of execution

To maximize speed, load your code as regular object code, with or without debugging information. You will not be able to do any run-time error checking, but you can do some source-level debugging—the amount depends on whether you load debugging information or not.

#6: Fastest startup

When startup and execution time are the most important considerations and run-time error checking and incremental turnaround times for changes are not important, start your project using **pdm** and load your code as an **a.out** file. This results in reasonably good debugging facilities and maximum speed of execution. Using **pdm** is especially useful for finding bugs quickly in existing code, rather than for debugging new code as you develop it.

Kinds of debugging supported

See Table 11 for a summary of the various kinds of debugging supported by ObjectCenter, the types of files you need to load for each kind, and the benefits provided by ObjectCenter's various debugging tools.

Table 11 Kinds of Debugging Supported by ObjectCenter

Debugging Activity	Kinds of Files	What ObjectCenter Does
Load-time error checking	source code	Issues errors or warnings for the following conditions:[ab] I/O errors Illegal characters Illegal constant formats Illegal escape sequences Lexical constant overflow Improper comments Preprocessing violations Macro expansion violations Syntax errors Illegal statements Illegal expressions Undefined identifiers Unused variables Improper type specifiers Illegal bitfield declarations Declaration violations Illegal parameter declarations Initialization violations Redefinition violations Linking violations Variable warnings
Run-time error checking	source code	Issues errors or warnings for the following conditions:[a] Undefined/questionable arithmetic operations Undefined/illegal pointer operations Enumerator warnings Losing information during conversions/assignments Function warnings Storage warnings

Table 11 Kinds of Debugging Supported by ObjectCenter (Continued)

Debugging Activity	Kinds of Files	What ObjectCenter Does
Run-time error checking	source code, regular object code, and instrumented object code[c]	Issues errors or warnings for the following conditions:[a] Memory allocation warnings Miscellaneous warnings (bad arguments to **strcpy**, **strcmp**, **bzero, longjmp**)
Run-time error checking	source code and instrumented object code[c]	Issues errors or warnings for the following conditions:[a] Using memory that has not been set Addressing errors (pointer dereference, alignment, array index errors)
Interactive source-level debugging	source code, object code, and **a.out** (with core or process)	Allows you to do all of the following: • Start your program under varying conditions that might affect its behavior • Stop your program on specified conditions • See what has happened when your program has stopped In component debugging mode, you can also easily change your program, so you can try out solutions to problems you discover.
Code visualization (Cross-Reference Browser)	source code and object code	Displays static cross-references for functions or variables in source and object files.
Code visualization (Data Browser)	source code, object code, and **a.out**	Allows you to examine values of data variables, including complex pointer structures.

a. Use the Manual Browser to see a list of messages in each category; issue the **man** command in the Workspace and select the “violations” topic.

b. Use the **config_c_parser** command to set the compiler configuration to be used for error checking.

c. See the **instrument** and **uninstrument** commands for more information.

Performance considerations

In this section we describe the various kinds of debugging activities shown in Table 11 in terms of performance. For an overview of the trade-offs between debugging techniques and some additional performance enhancements, see the **performance** entry on page 263.

Load-time error checking

Load-time error checking allows you to discover compile-time errors in your code as well as hazardous but legal usages of the C language.

If you want to use the load-time error checking features of ObjectCenter, you must load your code as source, even though this means your code will execute slowly. For load-time error checking, we recommend that you load in source form only those modules you are checking specifically for load-time errors; load as much of the rest of your code as possible in object form.

Run-time error checking

Run-time error checking allows you to discover errors and warnings related to memory allocation and usage as well as miscellaneous problems with enumerators, conversions, function calls, and storage.

As shown in Table 11, the specific run-time errors that ObjectCenter can find in your code depend on whether the code is loaded as source, regular object, or instrumented object code. By *instrumented* object code, we mean object code that has been modified so that it supports run-time error checking; *regular* object code has not been so modified. See the **instrument** entry on page 162 for more information about the errors reported for instrumented object code.

Regular object code that is linked and executed within ObjectCenter runs at nearly the full speed of the machine. Instrumented object code runs somewhat more slowly than regular object code, and source code runs much more slowly than either regular or instrumented object code.

So, for maximum speed with run-time error checking, we recommend that you load in source or instrumented object form only those modules you are checking specifically for run-time errors; load as much of the rest of your code as possible in regular object form.

Source-level debugging

Source-level debugging allows you to examine what is going on in a program while it executes. This kind of debugging is available in either of ObjectCenter's two modes, which are component debugging mode and process debugging mode. In component debugging mode, you load a program into ObjectCenter as any combination of source code, object code (instrumented or regular), and library files that you link and execute within ObjectCenter. In process debugging mode, you load your program as a fully linked executable.

For maximum speed of source-level debugging, load your files as regular object code in component debugging mode or as an **a.out** file in process debugging mode. In either case, your program will run at or near the full speed of the machine.

Differences in source-level debugging

The debugging features available in ObjectCenter vary somewhat according to whether you have loaded source, object, or **a.out** files.

Source-level debugging of a.out files

As previously mentioned, to perform source-level debugging with **a.out** files, you must use ObjectCenter's process debugging mode (**pdm**). Most ObjectCenter commands are exactly the same, whether or not you are in process debugging mode, but there are some differences. See the **pdm** entry on page 254 for more information.

Also, the debugging information in the symbol table in an **a.out** file varies according to whether or not you used the **-g** switch when you compiled the object modules that you linked to create it. See the **debug** entry on page 113 for more information about loading an **a.out** file.

Maximizing debugging capability with object code

You can load object code with and without debugging information. To get maximum debugging capability, load object files that have been compiled with the **-g** compiler switch.

Source-level debugging of source code vs. object code

ObjectCenter provides very similar source-level debugging capability for object code as for code loaded in source form, as long as the object code is loaded with debugging information. There are, however, a few differences.

If you load object files *with* debugging information:

- You cannot use the **stepout** command when you are stopped in object code.
- You cannot trace execution through object code as you can in source code.
- Some forms of the **action** command have no effect with code loaded in object form.

In addition, if you load object code *without* debugging information:

- You can stop on or set an action on a function name, but you cannot stop on or set an action on a particular line.
- You cannot step through the object code.

To examine some variables in object code, you must load the header file that defines them. To work around the following limitations, load a header file with the appropriate declarations, or swap one of the object files to source:

- You cannot get class and function template information from code loaded in object form.
- References are treated as if they are pointers.

Special considerations with C++ object files

Because of the way that C++ object files are generated, there are special considerations when examining some objects that are loaded in object form.

Setting breakpoints in inline functions

You can set breakpoints in inline functions loaded in object form only if the file was compiled using **CC**'s **+d** switch. The **+d** switch tells **CC** not to expand inline functions. (You can always set breakpoints in inline functions loaded in source form.)

Functions taking variable arguments

When **cfront** generates C files from C++ files for compilation, it discards ellipses in function definitions because some C compilers don't support them. Ellipses indicate that a function takes a variable number of arguments.

This means that if you call from source code such a function that is loaded in object form (with debugging information), ObjectCenter will issue an erroneous warning because the number of arguments does not match the number specified in the object file.

Because the violation is a warning (not an error), you can simply suppress it and continue.

Constants

When the C++ translator compiles a **const** object, it puts the compile-time value, not the identifier, in the object code. That means that you cannot examine **const** objects that are loaded in object form.

For example, say that the file **const**.**C** has the following definition:

```
const int i;
```

If that code is loaded in object form, you will get an error if you try to examine **i**:

```
-> load const.o
Loading: const.o
-> whatis i
'i' is undefined.
```

This is true even if the file is loaded with debugging information.

Enumerated types

If enumeration data types are loaded in object code that contains debugging information, you can examine the data type and variables of that type.

For example, say that **enum**.**C** has the following definition:

```
enum colors {red, green, blue} mycolor;
```

If that code is loaded in object form with debugging information, you can examine **colors**:

```
-> load enum.o
-> whatis colors
enum colors {
  red=0,
  green=1;
  blue=2
 };
```

When you examine a variable of an enumeration type, ObjectCenter tells you that it is an integer and returns the integer value of the enumeration:

```
-> mycolor;
(int) 0
```

You can specify the enumerators (members) if the object file is loaded with debugging information:

```
-> mycolor=blue;
(int) 2
```

However, if the object file is loaded *without* debugging information, you cannot examine the enumerated data type:

```
-> load -G enum.o
Loading: -G enum.o
-> whatis colors
'colors' is undefined.
```

You can examine variables of the enumerated type when they are in object code without debugging information, but you can only assign integral values to them; you cannot specify the enumerators:

```
-> mycolor;
(int) 0
-> mycolor=2;
(int) 2
-> mycolor=blue;
Error #718: blue undefined.
```

Functions that return **void**

When the C++ translator compiles functions that are defined as returning **void**, it changes their definition to functions returning **char** (it does this because not all C compilers accept **void** as a return type).

For example, say you have defined a function **foo()** in **enum.C** as follows:

```
void foo() { /* body */ }
```

If the file is loaded in object form, ObjectCenter will tell you the function returns **char**:

```
-> load enum.o
Loading: enum.o
-> whatis foo
char foo() /* defined */
```

Using object code without debugging information

When a call to an object code function without debugging information is displayed, the formal parameters for the function are not listed because they are not known by ObjectCenter. You can display the parameters by specifying a prototype for the object code function.

Debugging multiple processes

You can debug multiple processes in ObjectCenter. 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.

If the child process does an exec

If the child process does an **exec** and you want to debug the child program, you must modify the call to **exec...()** so that **objectcenter** is executed instead of the child program. Here is an example of a modified **exec**:

```
if (fork() == 0)
{
  printf("In the child\n");
#ifdef __OBJECTCENTER__
  execlp("objectcenter","objectcenter","-motif",0);
#else
  execlp("child_prog","child_prog",0);
#endif
}
```

After this code is executed, you can move the mouse to the new Workspace window, load the source code for the child program you want to debug, then run it.

NOTE You cannot use object code debugging for the child program in programs that fork; you can set breakpoints and step through code only in source code in the child process.

If your program fails to fork

In order to fork another ObjectCenter window, the **win_fork** option must be set to TRUE (the default). To check if **win_fork** is set, enter this command in the Workspace:

```
-> printopt win_fork
```

If **win_fork** is set to FALSE, enter this command to set it to true:

```
-> setopt win_fork
```

Failure to fork can also occur because there is not enough swap space for two copies of ObjectCenter.

If you are using **fork()** followed by **exec...()**, using **vfork()** instead will help. Here is a modification of the above code, which invokes Ascii ObjectCenter instead of the Motif version, as in the previous example:

```
if (fork() == 0)
{
  printf("In the child\n");
#ifdef __OBJECTCENTER__
  execlp("objectcenter","objectcenter","-ascii",
"-i" "/dev/ttyp9", "-o" "/dev/ttyp9",0);
#else
  execlp("child_prog","child_prog",0);
#endif
}
```

The arguments to **objectcenter** are different because a new window is not being created automatically. The arguments **-i** *device_name* and **-o** *device_name* name the devices that ObjectCenter will use for its input and output, respectively. Here, another terminal window was named to act as the console. Before doing this, you must get the correct name of the terminal window, then put it to sleep by issuing the following command:

```
% sleep 10000
```

delete

deletes debugging items

cdm	pdm
✔	✔

Command syntax

delete

delete all

delete *"file":line ...*

delete *number ...*

Description

<< *none* >>	Deletes all debugging items at the current break location. (**cdm** only)
all	Deletes all debugging items everywhere.
"file":line...	Deletes a breakpoint or action at the specified location. More than one location can be specified. (**cdm** only)
number...	Deletes the specified debugging item.

Usage

Use the **delete** command to delete a breakpoint, action, display, or trace.

To obtain the number of a debugging item, use the **status** command.

Zombied items

If **delete** is called on a debugging item currently active on the execution stack, the item will be *zombied* (marked for deletion) instead of being deleted immediately. A zombied item is deleted once it has completed executing.

See Also

action, **display**, **status**, **stop**, **trace**, **when**

demand-driven code generation

generating code only when used

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

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

Switches

See Table 12 for the switches that you can use to specify demand-driven code generation.

Table 12 Using the **-dd** Switch for Demand-Driven Code Generation

Switch Setting	What the Switch Tells ObjectCenter To Do
-dd=off	Generate all code, whether or not it is used; do not use demand-driven code generation.
-dd=on (the default setting)	Use demand-driven code generation exclusively. Generate only the code that is actually used in the module that is being loaded or compiled. In the case of functions, generate code for any definitions that might be used externally, even if they are not used in the particular module being loaded or compiled. In the case of classes, generate code only if the class is used in the file in which it is defined or if the class is declared with a forward reference. **-dd=on** is the default setting for loading C++ code into ObjectCenter and for compiling C++ code with ObjectCenter's version of the C++ translator.

Usage

Use the **-dd=on** and **-dd=off** switches in any of the following situations:

- With the **CC** command when compiling in a shell outside the ObjectCenter environment:

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

- With the **load** command in the Workspace:

  ```
  -> load -dd=off my_source.C
  ```

- In makefile target rules for generating object code from C++ source files:

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

- In makefile target rules used for loading a program into ObjectCenter:

  ```
          #load -C -dd=on ${C_OBJS}
  ```

Advantages of demand-driven code

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

In the first place, demand-driven code generation decreases the amount of debugging symbols that are generated by the C++ translator. This in turn reduces the size of object modules built by the **CC** language system. Generating fewer debugging symbols also means that the C++ translator produces a smaller C language file, so that compilation by the C compiler is faster.

Demand-driven code generation reduces the size of data structures built by ObjectCenter when you load source code into the programming environment or when you load object code that was created using demand-driven code generation.

Finally, using demand-driven code generation reduces the amount of executable code generated for inline functions when loading source code in the environment, or when compiling with **CC** outside the environment with **+d.** The **+d** switch to the **CC** command specifies that inline functions not be expanded inline.

TIP: Specifying demand-driven code generation as a project-wide property

If you are using the Motif or OPEN LOOK versions of ObjectCenter, you can specify demand-driven code generation as a project-wide property. See “Setting project-wide properties” in the *ObjectCenter User’s Guide.*

destruct

invokes destructors for static objects

cdm	pdm
✔	

Command syntax

destruct

destruct *file*

Description

<<*none*>>	Invokes static destructors for all loaded files.
file	Invokes static destructors for the specified file.

Usage

Use **destruct** to invoke destructors for static objects.

The **destruct** command does not affect objects created in the Workspace.

See Also

construct

detach

detaches from a running process

cdm	pdm
	✔

Command syntax **detach**

Description

<<*none*>> Detaches ObjectCenter from the running process that was attached using ObjectCenter's **attach** command.

Usage

Use the **detach** command to release a process from ObjectCenter's control. Detaching a process continues its execution.

After you use the **detach** command, a process is completely independent of ObjectCenter, and you can use **attach** with another process, or start a process with **run**.

NOTE If you leave process debugging mode or use the **run** command while you have an attached process, you kill that process.

See Also **attach**, **debug**, **pdm**

display

displays the value of a variable or expression

cdm	pdm
✔	✔

Command syntax

display *expression*

display *variable*

Description

expression — Ascii ObjectCenter: Evaluates the designated expression and displays its value whenever execution is stopped.

Motif and OPEN LOOK: Invokes the Data Browser, which creates a new display item each time you invoke the **display** command. The display item graphically displays the value of the variable or expression.

variable — Displays the value of the designated global or local variable whenever execution is stopped.

Options

The following ObjectCenter options affect the **display** command:

print_inherited — Filters the display of inherited data members.

print_pointer — Adds diagnostic information to pointer display.

print_runtime_type — Specifies pointer display as run-time rather than compile-time types.

print_static — Tells ObjectCenter to display static data members.

show_inheritance — Tells ObjectCenter to show full rather than truncated inheritance path.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **display** command to display the value of an expression or a variable. ObjectCenter displays the value whenever your program is stopped, including during single-stepping.

Local variables

The argument to **display** may contain references to local variables that are currently in scope. If execution later stops at a point where these variables are no longer in scope, the **display** will either generate an error (Workspace) or show the variable with the text in a dimmed or "greyed out" state (Motif or OPEN LOOK).

To explicitly display a particular instance of a variable in component debugging mode, use the scoping syntax to qualify the name. In the example below, variable **node** is both a global and a local variable. Qualifying the name **node** by the function indicates that the display should act upon the local instance.

```
C++ -> display func 'node
```

See 'Specifying a variable's location' on page 426 for more information about this syntax.

Manipulating display items

Display items can be deleted with the **delete** command and examined with the **status** command.

See Also

delete, **dump**, **info**, **print**, **status**, **whatis**, **whereis**

down

moves down the execution stack

cdm	pdm
✔	✔

Command syntax

down

down *number*

Description

<< *none* >>	Moves the current scope location down one level on the execution stack. Motif and OPEN LOOK: Source panel shows file scoped to location and highlights it with an arrow.
number	Moves the current scope location the specified number of levels down on the execution stack.

Usage

Use the **down** command to move the current scope location down the execution stack, away from the top level of the Workspace and toward the current break level.

The scope location is the point at which all variables, types, and macros are scoped. When a break level is generated, the scope location is set to the point at which execution was interrupted.

When at a break level, the **where** command can be used to display the execution stack. The **whereami** command can be used to display the break location and the current scope location.

See Also

cont, **reset**, **up**, **where**, **whereami**

dump

displays all local variables

cdm	pdm
✔	✔

Command syntax

dump

dump *function*

dump *text*

Description

<< *none* >>	Displays the name and value of each variable local to the current scope location.
function	Displays the name and value of each variable local to the specified function.
text	Displays the name and value of each variable contained in an arbitrary text string.

Usage

Use the **dump c**ommand to display the names and values of local variables.

Typically you use the *text* argument by selecting a range of text, then issuing the **dump** command. ObjectCenter displays the name and value of each of the variables contained in the text string.

See Also

display, **info**, **print**, **whatis**, **whereis**

edit

invokes your editor at a specified location

cdm	pdm
✔	✔

Command syntax

edit
edit *identifier*
edit *file*
edit *"file":line*
edit *function*
edit *line number*
edit workspace

Description

<< *none* >>	Loads the current file into your editor, positioned at the current list location.
identifier	Loads the file containing the defining instance for the identifier into your editor, positioned at the location of the definition. (The identifier can be a variable, typedef, macro, or class / struct / union tag. If the defining instance is ambiguous, nothing is loaded into the editor.)
file	Loads the specified file into your editor, positioned at the top of the file.
"file":line	Loads the specified file into your editor, positioned at the specified line in the file.
function	Loads the file containing the specified function definition into your editor, positioned at the start of the function.

line number	Loads the file specified by the current list location into your editor, positioned at the specified line number.
workspace	Appends all workspace definitions to a file and invokes your editor on the file.

Options

The following ObjectCenter options affect the **edit** command:

cxx_suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.
editor	(Ascii ObjectCenter only) By default this option is unset. Set it only if there is no edit server in your environment. Possible values are **vi** and **emacs**.
path	Specifies the search path for editing files.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **edit** command to facilitate quick debug-edit-run turnaround times by invoking your editor (specified by the **editor** option) to edit a file at a specified location. In all cases, once the editor is invoked, the current list location is set to the file and line number edited.

Use the **edit workspace** command to save code you define in the Workspace. 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 more information, see 'Using the edit workspace command' on page 433.

See Also

edit server

edit server

An edit server is a software utility that allows you to attach an editor to ObjectCenter so that, whenever you issue an edit request, the server automatically invokes an editing session using the attached editor.

ObjectCenter provides built-in edit servers for **vi** and FSF GNU Emacs. For information about using GNU Emacs with ObjectCenter, see the **emacs integration** entry on page 141.

emacs integration

ObjectCenter provides two ways to integrate ObjectCenter with GNU Emacs: You can connect your GNU Emacs session to ObjectCenter so that your Emacs session is used when you use the **edit** command or select an Edit symbol or button. If you use FSF GNU Emacs 19 or a later version, you can also invoke ObjectCenter from within your Emacs session and use the Emacs Main Window.

A compatible version of FSF GNU Emacs is available by anonymous ftp from the host **ftp.centerline.com** in the **/pub/TOOLS/emacs** directory. For more information, refer to the **README** file in that directory.

To use either of these features, you must load **clipc.el**. You can do this by adding the following lines of ELISP code to your **.emacs** startup file after any existing **load-path** lines:

```
(setq load-path (cons "path/CenterLine/lib/lisp" load-path))
(load "clipc")
```

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

For example, you might have these lines at the beginning of your **.emacs** file:

```
(setq load-path (cons "/usr/local/emacs/local-lisp" load-path))
(setq load-path (cons "install_path/CenterLine/lib/lisp" load-path))
(load "clipc")
```

Connecting GNU Emacs to ObjectCenter

Once you have these lines of ELISP in your **.emacs** file, you need to have Emacs load them in your current Emacs session. If you already have an Emacs session running that you want to connect to ObjectCenter, then select these lines of ELISP and evaluate the region in Emacs:

```
M-x eval-region RET
```

Alternatively, you can load the file **clipc** with the following Emacs commands:

```
M-x load-file RET clipc RET
```

If you do not have an Emacs session running, invoke Emacs and these lines will be read with the .emacs file. Once Emacs has loaded the new lines of ELISP, you establish a connection to ObjectCenter by using this Emacs command:

```
M-x cl-edit RET
```

Emacs Main Window

If you use FSF GNU Emacs version 19 or a later version, you can invoke ObjectCenter from within Emacs.

Use the Emacs command **M-x objectcenter** to start your ObjectCenter session. Emacs prompts

```
"Run ObjectCenter (like this): path-to/objectcenter"
```

where *path-to* is the path to your **objectcenter** executable. You can edit this path if it does not show the executable you want to use. Press Return to start ObjectCenter in component debugging mode, or enter the **-pdm** switch to start in process debugging mode. You can also give the name of a project file as an argument.

ObjectCenter starts up in a new buffer called **cl-workspace**. All the menus at the top of the Emacs window are replaced with the menus from ObjectCenter's Main Window, except the In/Out and Help menus.

You can use most of the commands and features available in ObjectCenter to prototype and debug code in this buffer.

Source window

When you load your application into ObjectCenter, run it, and stop at a breakpoint or error, the source code is displayed in a separate buffer above the Workspace. An arrow (=>) indicates the line at which execution stopped. As you step through your code, a new buffer is used for each file you step through. You can edit your code directly in the source window.

Button Panel

The Button Panel is available in a separate window. To open it, select Button Panel from the Browsers menu.

Setting breakpoints, tracepoints, and actions

You can set a breakpoint on a line of the code displayed in the source window, or remove an existing breakpoint, by holding down the Control key and clicking the left mouse button anywhere on the line. Lines with breakpoints set on them display in reverse video.

You can also use the stop, trace, and action workspace commands or selections from the Debug menu. Debug menu selections that require an argument bring up a prompt in the minibuffer. Set Action brings up

a new buffer in which you enter the action with one statement on each line. Type Control-c Control-c to save the buffer and set the action.

Selecting text

To select text, hold down the Left mouse button and drag over the item you want to select as in ObjectCenter. You can then select an item from the Examine menu or the popup Expression options menu. Press Shift plus the Right mouse button to display the Expression options menu.

Key bindings

You can use the following key bindings, as well as others available in Emacs:

Ctrl-c Ctrl-b	**build**
Ctrl-c Ctrl-d	**display**
Ctrl-c Ctrl-n	**next**
Ctrl-c Ctrl-r	**run**
Ctrl-c Ctrl-s	**step**
Ctrl-c Ctrl-c	interrupt execution
Tab	complete filename
Meta-?	list possible completions
Esc-p	Scroll backwards through input history
Esc-n	Scroll forwards through input history

Limitations

Some features that are available in ObjectCenter's Graphical User Interface are not available in the Emacs Main Window:

- Line numbers, breakpoint symbols, and the scope arrow in the source window. We show breakpoints in reverse video.
- User Defined and Button Panel items on the ObjectCenter menu.
- The Error Browser button.

Some features described in the **Workspace** entry are not available or work differently. You can repeat the previous line of input with ##, or the *n*th. previous line with #-*n*, but the expansion syntax for #$, #*, and #: described on page 422 is not available. You cannot use the <ESC><ESC> and <ESC>x sequences for command, name, and filename completion described on page 424.

email

sends electronic mail to CenterLine Software

cdm	pdm
✔	✔

Command syntax

email

email *file*

Description

<< *none* >>	Ascii ObjectCenter: Invokes the UNIX **mail(1)** electronic mail utility. Motif and OPEN LOOK: Opens the **email** dialog box.
file	Ascii ObjectCenter: Invokes the UNIX **mail(1)** electronic mail utility, sending the contents of the specified bug report file or suggestion file to CenterLine Software.

Options

The following ObjectCenter option affects the **email** command:

email_address Specifies the electronic mail address for the **email** command.

The option is set in the following file:

CenterLine/configs/support-defs

By default the value is as follows:

objectcenter_support@centerline.com

NOTE In the Motif and OPEN LOOK versions, the **email_address** option has no effect.

See the **options** entry for more details about each option. CodeCenter does not support this option in process debugging mode (**pdm**).

Usage

To report bugs or offer suggestions, use the **email** command to send an electronic mail message to CenterLine Software. When you send a bug report, include examples of the source code that produced the problem, if possible.

When you issue the **email** command, you can use the UNIX **mail(1)** electronic mail utility's escape sequences.

english

describes a C type in English

cdm	pdm
✔	

Command syntax

english *type_expression*

english *identifier*

Description

type_expression	Displays a prose description for the type of the specified type expression.
identifier	Displays a prose description for the type of the specified identifier (variable, function, class / struct / union tag, or typedef).

Usage

Use the **english** command to clarify a C type expression or to display a prose description of the type of an identifier.

Example

The following example indicates both ways **english** can be used to describe a type:

```
-> char *(*func)();
-> english func
pointer to function returning pointer to char.
-> english char *(*)()
pointer to function returning pointer to char.
```

Restrictions

The **english** command is available only in C mode.

See Also

help, **man**

environment variables

What is an environment variable?

The environment of a process is an array of strings; each string is called an environment variable. By convention, each string, or environment variable, has the following form:

NAME =[*value*]

For instance, the following string is an environment variable:

EDITOR=vi

The shell makes these strings available to programs through **envp,** which it passes as the third argument to **main()** whenever a program begins execution. See the UNIX manual pages for **environ** and **execv** for more details.

When you are in a C shell, you can manipulate environment variables for programs in that shell by using the **csh** built-in commands **setenv** and **unsetenv** along with the **printenv** shell command. See the UNIX manual pages for **csh** and **printenv** for more details about these commands.

Setting and examining environment variables

Similarly, when you are in the ObjectCenter environment, you can use ObjectCenter's **printenv**, **setenv**, and **unsetenv** commands to manipulate environment variables for programs within ObjectCenter:

printenv	Displays the values of environment variables
setenv	Sets the values of environment variables
unsetenv	Unsets environment variables

See the reference pages for **printenv**, **setenv**, and **unsetenv** for more details about these commands.

These ObjectCenter commands affect only the environment variables for the program you are examining in ObjectCenter. They do not affect the environment variables used by ObjectCenter to control its own operation, nor do they affect the value of environment variables outside of ObjectCenter. To control ObjectCenter's operation, use ObjectCenter's options.

Environment variables used by ObjectCenter

For instance, changing the **EDITOR, DISPLAY**, or **PAGER** shell variables with ObjectCenter's **setenv** command does not affect which editor, display screen, or paging program ObjectCenter uses. To modify ObjectCenter's behavior, use ObjectCenter's **setopt** command with the appropriate option.

Environment variables in makefiles

You should be careful about changing the values for any environment variables that you use in a makefile. If you change the value for an environment variable from within ObjectCenter, you have not changed its value in the process that is invoked when you use the **make** command to recompile your program. This means that you might not get the results you intend when the **make** evaluates the environment variable.

Expanding environment variables in ObjectCenter

Use the **#$** syntax described in Table 32 on page 422 to expand environment variables.

CenterLine and ObjectCenter environment variables

By convention, environment variables that are specific to the ObjectCenter product have the following prefix:

OBJECTCENTER_

Similarly, environment variables that are specific to all CenterLine products have the following prefix:

CENTERLINE_

For example, ObjectCenter normally displays a message when linking from a library:

```
Linking from ... Linking completed.
```

You can suppress the linking messages by setting the environment variable **CENTERLINE_LINK_SILENT** before starting ObjectCenter. This is particularly useful in Ascii ObjectCenter when linking from shared libraries: run-time linking messages will not obscure your program's output.

See Also

printenv, **setenv**, **setopt**, **unsetenv**, **unsetopt**

expand

lists the functions that could be called by a C++ statement

cdm	pdm
✔	

Command syntax

expand *simple_statement*

expand *compound_statement*

Description

simple_statement Lists the functions that the specified statement could call, including user-defined conversions.

compound_statement Lists the functions that the specified compound statement (a block of code delimited by braces, which forms a single statement) could call, including user-defined conversions.

Usage

Use the **expand** command to clarify which functions could possibly be called if a block of C++ code is executed. (The set of functions *actually* called during a given run might depend on run-time conditions and so might be a subset of the functions listed by **expand**.) Expanding a statement allows you to see implicit function calls and disambiguates overloaded functions and operators.

Showing implicitly called destructors

If any class objects are constructed in the code you give as an argument to the **expand** command, **expand** shows the destructors that would be called when the objects go out of scope. These implicitly called destructors are shown at the end of the listing of the functions called.

The following example shows the **expand** command disambiguating an overloaded operator:

```
-> load String.C
Loading (C++): String.C
-> String s1="Object";
(class String *) 0x2d4590 /* (class String) s1 */
-> String s2="Center";
(class String *) 0x2d47d0 /* (class String) s2 */
-> expand s1+s2
String operator +(const String &, const String &)
String::~String()
```

Restrictions

When you issue **expand** in the Workspace, variables must currently be in scope. This restriction does not apply to using **expand** from the GUI.

See Also

print, **whatis**

fg

returns to ObjectCenter

cdm	pdm
✔	

Command syntax **fg**

Description

<< *none* >>	Returns to ObjectCenter after a **suspend** command. (Ascii ObjectCenter only)

Usage Use the **fg** command to return to ObjectCenter after being suspended.

See Also **quit**, **save**

file

displays and sets the current list location

cdm	pdm
✔	

Command syntax

file

file *filename*

Description

<< *none* >>	Displays the name of the file containing the current list location.
filename	Sets the current list location to the top of the specified file.

Usage

Use the **file** command to display and set the current list location. Commands such as **action**, **edit**, **list**, and **stop** use the list location as the default location unless specifically overridden by an argument.

The **file** command changes which static variables are visible at the top level in the Workspace. Another way to view a multi-defined static variable is to preface the variable name with the function or filename in which it is defined (for example**, `file.c`variable** or **func`variable**).

See Also

action, **edit**, **list**, **stop**

gdb

executes a **gdb** command

cdm	pdm
	✔

Command syntax

gdb *gdb_command* **[** *argument* **]** ...

Description

gdb_command [*argument*] Executes *gdb_command* [*argument*] as if it were typed to a **gdb** command prompt.

Usage

The **gdb** command allows you to stay in process debugging mode and execute **gdb** commands. For instance, the following invokes **break**, a **gdb** command, with **20** as the argument:

```
(pdm) 1 -> gdb break 20
```

NOTE Although we provide access to native **gdb** commands as a convenience, we do not provide any additional support for native **gdb** commands**.**

For more information on **gdb** commands, you can use the **gdb help** command:

```
(pdm) 1 -> gdb help
```

Documentation on **gdb** is available from CenterLine by using anonymous **ftp**. For information, refer to 'Distribution' on page iii.

See Also

gdb_mode

gdb_mode

changes from **pdm** mode to **gdb** mode

cdm	pdm
	✔

Command syntax **gdb_mode**

Description

<< *none* >> Changes from **pdm** mode to **gdb** mode.

Usage

Use the **gdb_mode** command when you want to issue a series of **gdb** commands without prefacing every command with the **gdb** command.

To use **gdb** along with **pdm**, issue the **gdb_mode** command in the ObjectCenter Workspace while you are in process debugging mode:

```
(pdm) 1 -> gdb_mode
(gdb)
```

Once you are in **gdb** mode, you can use *only* the **gdb** command set:

```
(gdb) break 20
(gdb) when
Undefined command: "when". Try "help".
```

You can get back to process debugging mode by typing the following command:

```
(gdb) pdm
(pdm) 2 ->
```

NOTE Although we provide the **gdb_mode** command as a convenience, we do not provide any technical support for **gdb.**

For more information on **gdb** commands, you can use the **help** command while in **gdb** mode.

Documentation on **gdb** is available from CenterLine by using anonymous **ftp**. For information, refer to 'Distribution' on page iii.

See Also **gdb**

help

displays usage information about commands

cdm	pdm
✔	✔

Command syntax

help

help *command*

Description

<< *none* >>	Lists the names of ObjectCenter commands by category.
command	Displays a summary of syntax and usage information for the specified command.

Usage

Use the **help** command for quick online help for ObjectCenter commands.

See Also

english, **man**

history

lists previously entered input

cdm	pdm
✔	✔

Command syntax

history

history *number*

Description

<< *none* >>	Displays all input lines previously entered from the Workspace.
number	Displays the specified number of input lines entered from the Workspace.

Options

The following ObjectCenter option affects the **history** command:

line_edit	Adds line editing, command completion, and extensive history capabilities to the Workspace.

Usage

Use the **history** command for easy recall of previously issued commands and to monitor the debugging sequence leading to a given state.

Use **##** to repeat the immediately previous command, and use **#***history_line_number* to repeat the command specified by *history_line_number*.

Pressing Control-p scrolls backward through the history list. Pressing Control-n scrolls forward through the history list.

To save the list of input lines entered from the Workspace in a file, use the following command:

```
-> history #> file_name
```

See Also

Workspace

ignore

allows signals to pass directly to the program

cdm	pdm
✔	✔

Command syntax

ignore

ignore *signal-name*

ignore *signal-number*

Description

<< *none* >>	Lists the unprefixed name of the signals that are currently ignored.
signal-name	Disables trapping for the designated signal, allowing the signal to pass directly to the program, which can execute a signal handler if it has been specified.
signal-number	Disables trapping for the designated signal, allowing the signal to pass directly to the program, which can execute a signal handler if it has been specified.

Usage

Use the **ignore** command for any signal that you want to pass directly to the program. Once an ignored signal is passed to the program, the program executes any signal handlers specified for it.

Signal numbers

To obtain the number for a signal, consult the UNIX reference manuals for your system.

Signal names

With the **ignore** command, the signal name can be in uppercase or lowercase letters, and it can be used with or without the prefix “SIG”. For example, the following commands are equivalent:

```
-> ignore SIGHUP
-> ignore sighup
-> ignore HUP
-> ignore hup
```

Signals ignored

To obtain a list of signals ignored on your platform, type the **ignore** command without any arguments.

NOTE Even if a signal is ignored, it interrupts system calls, such as **select()**, that are interruptible.

Restrictions

When a signal is caught and a break level is generated, the signal is consumed. Ignoring the signal at the break level and continuing execution will not regenerate the signal and pass it to the program.

Control-z at the command prompt is not interfered with (Ascii ObjectCenter only).

Control-z during execution or in the run window is always handled as a signal-deliver, generating an error if not trapped by the user program.

Ignoring SIGINT causes SIGQUIT to perform interruption duties. Ignoring both of them interferes with stopping execution.

The signals SIGTTIN and SIGTTOU will never suspend execution; if not trapped and ignored they will generate an error.

When an **exec**() is done within ObjectCenter, the inherited signal mask only includes signals that have been ignored. Also, the SIGQUIT, SIGTRAP, and SIGEMT signals are never present in the inherited signal mask (**cdm** only).

See Also

catch

info

displays information (address, name, size, and type) for data at a specific memory location

cdm	pdm
✔	

Command syntax

info *address*

info *lvalue*

info *variable*

Description

address	Displays information for the data at *address*, which must be a hexadecimal value.
lvalue	Evaluates *lvalue* and uses the result as an address. Displays information for the data at the evaluated address.
variable	Displays information for the data at the address named by a variable.

Usage

Use the **info** command to display information about the data located at a specific address. This command is especially useful for determining what a pointer points to.

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. If the address is being watched by a debugging action or the address contains a bad pointer, then **info** also indicates this.

Example

In the example below, **info** is used to display information about a value stored in a pointer.

```
-> int i, *ptr;
-> ptr = &i;
(int *) 0x125278 /* i */
->
-> info ptr
address = 0x134662, name = ptr
Size = 4, contains type: pointer
->
-> info *ptr
address = 0x125278, name = i
Size = 4, contains type: int
->
-> info 0x125278
address = 0x125278, name = i
Size = 4, contains type: int
->
```

See Also

display, **dump**, **whatis**, **whereis**

instrument

enables run-time error checking for an object file

cdm	pdm
✔	

Command syntax

instrument

instrument *file* **...**

instrument all

Description

<< *none* >>	Lists names of instrumented files
file ...	Adds information to *file,* making it possible for ObjectCenter to perform certain kinds of run-time error checking. The *file* argument can be the name of any loaded object file or any loaded source file that might be swapped to an object file. If *file* is a source file that is swapped using the **swap** command, ObjectCenter automatically instruments the object file when it swaps it with the source file.
all	Instruments all object files currently loaded. This includes modules linked in from libraries.

Options

If you want ObjectCenter to instrument your files automatically as they are loaded, set the **instrument_all** option.

The following options affect the **instrument** command:

instrument_byte	Checks for unset memory that is used one byte at a time.
instrument_space	Allocates space for instrumented code according to the value of this option. By default, the option is set to **2**, which requires an amount of space equal to approximately half the text size of your application. If you set this option to **0**, you save space, but you cannot instrument any object code.

unset_value Use this value to detect memory that has not been set. Every byte of memory allocated by the **malloc()** functions, as well as memory for automatic variables, is set to the value of **unset_value**. If this option is set to **0** (**setopt unset_value 0**), ObjectCenter no longer diagnoses that a variable is used without being set.

Using the instrument_byte option

By default, ObjectCenter fails to report an error for usage of uninitialized memory when your code accesses such memory one byte at a time. For instance, consider the following example:

```
char *cp, *dp;
cp=malloc(10);
dp=malloc(20);
for(i=0; i<10; i++)
   dp[i]=cp[i];
/* cp has not been initialized, but
   ObjectCenter uses it anyway and does
   not report memory as used before set */
```

The **for** statement in the preceding code causes the system to obtain one byte of memory at a time to acquire the value of **cp[i]**. In this situation ObjectCenter does not report the programming error, namely the use of **cp** without initializing its value, unless you set the **instrument_byte** option.

Usage

Use the **instrument** and **uninstrument** commands to enable and disable run-time error checking of loaded object code. Enabling the run-time error checking of loaded object code is called *instrumenting* the file.

NOTE If you are using the Motif or OPEN LOOK version of ObjectCenter, you can use the Project-Wide Properties window to control instrumentation of your object files. This window is accessible from the **Project Properties** selection off the Project pulldown menu in the Project Browser.

Kinds of errors reported

ObjectCenter can report the following kinds of errors in instrumented object code:

pointer bounds errors	ObjectCenter checks pointer bounds for global data and all memory allocated with **malloc()**. Note that pointer bounds checking is not performed for pointers to automatic variables—that is, for pointers pointing into the stack.
accessing uninitialized memory	ObjectCenter generates "used before set" messages; they report memory allocated with **malloc()** that is used by your program without being initialized. By default, ObjectCenter checks for uninitialized memory on 2-, 4-, and 8- byte memory references. Note that this type of checking is not performed for pointers to automatic variables—that is, for pointers pointing into the stack.

Instrumenting static libraries

You can use the **instrument** command to instrument an object module in a static library by using the following syntax:

instrument *library_pathname/object_module_name*

The module must be linked to your program before you instrument it; you can accomplish this without running your program by issuing the **link** command.

Run-time error checking in source or object code

ObjectCenter allows you to perform automatic run-time error checking on your program with any combination of source code and instrumented object code. Your decision about how to load your program for run-time error checking depends on how you want to balance the trade-offs of speed versus completeness of error checking.

Performance considerations

Keep in mind the following facts about ObjectCenter:

- Source code executed within ObjectCenter runs more slowly than either instrumented object code or regular object code.
- Instrumented object code executed within ObjectCenter runs faster than source code but slower than regular object code.
- Regular object code executed within ObjectCenter runs significantly faster than source code and somewhat faster than

instrumented object code. Regular object code runs at the full speed of the machine.

Unless you want to save memory or loading time, we recommend that you use the **instrument** command with object code that was compiled with debugging information (that is, with the **-g** option). That way you can shift easily to other source-level debugging techniques to trace and correct any run-time errors that are reported.

See the "Loading source versus object code versus executables" section on page 116 for more discussion of trade-offs in the way you load your program.

Errors detectable in source but not in object code

ObjectCenter can detect some errors in programs loaded in source form that it does not detect in object code, even if you instrument the object code. Here are some examples.

Some errors in pointer arithmetic

ObjectCenter can detect pointers that are slightly out of bounds in both source and instrumented object code, but sometimes it can detect pointers that are significantly out of bounds only in source code.

Suppose you intend to add **62** but you actually add **622** to a pointer address. The resulting pointer is significantly out of bounds for the variable you declared it to reference, but it happens to contain a legal address. If you load your code as source, ObjectCenter generates a warning about the pointer being out of bounds. In contrast, if you load your code as object code (instrumented or regular), ObjectCenter may not generate any message about the pointer, even though it is no longer considered valid.

Array index errors

ObjectCenter detects array index errors in source code, but it cannot always detect them in instrumented object code. For example, suppose you declare **s1** as follows:

```
struct S {
          int a;
          char b[4];
          int c;} s1;
```

Then you make the following assignment:

```
s1.b[5] = 0; /* this field was declared as b[4] */
```

If this code is loaded as source, ObjectCenter issues a warning for the out-of-bounds array index, **s1.b[5]**. In contrast, if this code is loaded as instrumented object code, ObjectCenter does not report an error and allows the unintentional assignment of **0** to the field **s1.c**.

Restrictions

This section lists the known restrictions for instrumenting object code.

Spurious warnings

If your code causes a bit pattern in memory to match the bit pattern represented by the value of the **unset_value** option, ObjectCenter generates spurious "used before set" warnings for your code. This is not likely to happen. However, if it does happen, you can work around this problem by doing one of the following:

- Suppress the warning
- Change the **unset_value** option to another value

ObjectCenter sometimes generates spurious warnings in connection with copies of structures; if you get a "used before set" message on a line of code containing a copy of a structure, you can probably ignore the warning.

Shared libraries

You cannot instrument object code contained in shared libraries.

Large object files

Some object files may be too large to instrument. If you get a message telling you that your object file is too large, we recommend that you divide the source code into smaller modules and compile them, creating more object modules that are smaller than the one that is too large.

NOTE Refer to the *ObjectCenter Platform Guide* for possible additional information about **instrument** that is specific to your platform.

See Also

debugging, **uninstrument**

keybind

changes bindings used by the in-line editor in the Workspace

cdm	pdm
✔	

Command syntax

keybind

keybind *key*

keybind *key key_comd args*

keybind *key key_function*

Description

<< *none* >>	Displays the current key bindings and all functions that can be bound to keys.
key	Displays the key function that the specified key sequence (*key*) is currently bound to.
key key_comd args	Binds the specified key sequence (*key*) to the specified key command (*key_comd*), along with the specified arguments (*args*) that are passed to the Workspace command line when the key command is called. See Table 13 for a list of values for *key_cmd*.
key key_function	Binds the specified key sequence (*key*) to the specified key function (*key_function*). See Table 14 for a list of values for *key_function*.

Options

The following ObjectCenter options affect the **keybind** command:

eight_bit	Tells ObjectCenter to treat input and output as 8-bit characters.
line_edit	Adds line editing, command completion, and extensive history capabilities to the Workspace.
line_meta	Lets all 8 bits pass as input.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **keybind** command to customize the key bindings for in-line editing in the Workspace. The default bindings are designed to mimic the editing commands used with **emacs** and **tcsh**.

NOTE The key sequences used for in-line editing are usually control characters or escape sequences. When specifying these key sequences to **keybind**, you must quote the sequence so that ObjectCenter does not invoke its current binding. You can quote a control or escape sequence by prefacing it with **Control-v**, which is bound to the **quote** function.

Key commands

Use the following syntax for binding commands to keys:

keybind *key key_comd args*

where *key_comd args* is one of the values shown in Table 13.

Table 13 Commands as Arguments for the **keybind** Command

Command	Description
shell *args*	Executes a subshell with *args* as the arguments.The output of the subshell is displayed on the screen. The name of the subshell to start is taken from the **subshell** option.
user *args*	Executes a subshell with *args* as the arguments. The name of the subshell to start is taken from the **subshell** option. The current line of input is sent as the input stream. The output of the subshell replaces the current line. This option is useful for adding a preprocessor that translates a line of input. The example below binds the key **Control-m** so that it will send the current line to **m4 macro_files**, with the resulting output substituted for the current line: `C++-> keybind ^V^M user m4 macro_files` Note that the character ^**V** was used to prevent interpretation of the ^**M** character.

Table 13 Commands as Arguments for the **keybind** Command (Continued)

Command	Description

command *args*	Executes the command *args* in a subshell with the current line passed as the arguments to the command. The name of the subshell to start is taken from the **subshell** option. The binding for the Esc-x key illustrates how this is done for the command **echo**:
	`/* The shell executes echo */`
	`/* load *.c and the result */`
	`/* is redisplayed */`
	`C++ -> load *.c Esc-x`
	`C++ -> load test.c foo.c bar.c`

macro *args*	Inserts *args* into the current line with full interpretation of all special characters. The example below binds the key Control-l to echo the string **load** .**c**; the Control-b characters move the cursor back before the suffix .**c**. `C++-> keybind ^V^L macro load .c^V^B^V^B` Note that the character ^**V** was used to prevent interpretation of the control character ^**B**.
alias *args*	Inserts *args* into the current line with no interpretation of special characters.

Key functions

Use the following syntax for binding keys used for in-line editing:

keybind *key key_function*

where *key_function* is one of the functions listed in Table 14.

Functions that perform an operation on a word, such as **word_delete_prev**, recognize any legal C++ or C identifier as a word. A legal C++ or C identifier is a combination of alphanumeric characters including the **_** and **$** characters.

NOTE For control-key sequences, press the Control key and the letter at the same time; for escape sequences, press and release the Escape key, then press another key.

Table 14 Key Functions Available for the **keybind** Command

Behavior Affected	Key Function	Default Key Binding	Description
Cursor movement	**beginning_of_line**	Control-a	Moves the cursor to the beginning of the line.
	backward_char	Control-b	Non-destructive backspace.
	end_of_line	Control-e	Moves the cursor to the end of the line.
	forward_char	Control-f	Moves the cursor forward one character.
	backward_word	Esc-b	Non-destructive backspace over the previous word.
	forward_word	Esc-f	Moves the cursor past the next word.
	reverse_search	Esc-r *char*	Searches backward for char. If char is **r**, then searches for the same character as the previous **reverse_search** or **forward_search**. To search for the character **r**, **reverse_search** must be bound to another escaped letter.
	forward_search	Esc-s *char*	Searches forward for char. If char is **s**, then searches for the same character as the previous **reverse_search** or **forward_search**. To search for the character **s**, **forward_search** must be bound to another escaped letter.
Deleting text	**delete_or_complete**	Control-d	Deletes the character under the cursor or performs identifier completion if at the end of the line. If the word under the cursor could refer to several identifiers, the unambiguous portion is completed, and all possible identifiers are displayed. If the cursor is at the beginning of a line, a **Control-d** is echoed, causing execution to continue if ObjectCenter was at a break level.

Table 14 Key Functions Available for the **keybind** Command (Continued)

Behavior Affected	Key Function	Default Key Binding	Description
	delete_backward	Control-h	Destructive backspace.
	delete_search	Esc-K *char*	Deletes characters from the current cursor position until character char.
	delete_to_end	Control-k	Deletes characters from the current cursor position until the end of the line.
	kill_line	Control-u	Erases the line.
	word_delete_prev	Control-w	Deletes the previous word.
	word_delete_next	Esc-d	Deletes the next word.
Inserting text	**(Control-@)**	Esc-y	Marks a line in the history list for later yanking with the **history_yank** command.
	tab	Control-i	Inserts spaces until the next tab stop. This also expands any history invocations that were just entered.
	next_history	Control-n	Edits the next (more recent) history line.
	previous_history	Control-p	Edits the previous (less recent) history line. If the previous line contains the same text as the current line, it is skipped. If the current line contains some text before this function is invoked, then only previous lines that begin with this text pattern are displayed.
	transpose_chars	Control-t	Transposes the two characters preceding the cursor.

Table 14 Key Functions Available for the **keybind** Command (Continued)

Behavior Affected	Key Function	Default Key Binding	Description
	quote	Control-v *char*	Inserts char without any key mapping. This is used to insert control characters.
	yank	Control-y	Inserts into the current cursor position the text deleted by the most recently performed **delete_search**, **word_delete_next**, or **word_delete_prev**.
	beginning_of_history	Esc-a	Edits the first line in the history list.
	end_of_history	Esc-e	Edits the last line in the history list.
	history_yank	Esc-y	Inserts the history line marked with **set_mark** (**Control-@**) into the current line.
	complete	Esc-Esc	Complete the name under the cursor. This is similar to pressing Control-d. If the word under the cursor could refer to several identifiers, the unambiguous portion is completed, and all possible completions are displayed.
Information	**explain**	Control-x	Prints the definition of the C++ or C identifier located under the cursor.
	help	Esc-h	Displays help information for the command located under the cursor. If the cursor is located at the beginning of a blank line, then summary help information is displayed.
	man	Esc-m	Displays the manual page for the command located under the cursor. If the cursor is located at the beginning of a blank line, then a summary manual page is displayed.

Table 14 Key Functions Available for the **keybind** Command (Continued)

Behavior Affected	Key Function	Default Key Binding	Description
Miscellaneous	**interrupt**	Control-c	Interrupts reading this line of input. The entire line buffer is flushed.
	reset	Control-g	Resets the state of the line editor.
	execute	Control-j	Executes this line.
	clear_screen	Control-l	Clears the screen.
	execute	Control-m	Executes this line.
	correct_typo	Control-o	Tries to make sense of previous line of input.
	redisplay	Control-r	Redisplays the current line.
	suspend	Control-z	Suspends ObjectCenter and returns to the shell.
	quit	Control-\	Quits ObjectCenter.
	prefix	Esc	Invokes a multi-character key binding.
	multi_prefix	Esc-[	More complicated key bindings, which are usually used for arrow and function keys.
	number	Esc-n	Repeats the next command four times. This is effective for most cursor movement functions, delete functions, and search functions.
	undo	Esc-u	Undoes the last non-trivial change.
	command echo	Esc-x	Expands wildcards or shell variables in the current line of input by sending them through /**bin**/**sh**. If the line contains a redirection symbol, the expanded output will get redirected by the shell.

Table 14 Key Functions Available for the **keybind** Command (Continued)

Behavior Affected	Key Function	Default Key Binding	Description
	space	space	Inserts a space at the current cursor position. Any history invocations are expanded.
	eof	eof	Sends an **end-of-file**.
	self_insert	self_insert	Inserts this character.
	bad	bad	Rings the bell and does not echo the character.

Arrow key functions

See Table 15 for a list of arrow keys and their default key functions. The arrow keys on most keyboards, including Sun, DEC™ Microvax, and standard VT™-100 compatible terminals, are supported with these functions. Also, the default bindings conform to the ANSI standard escape sequences.

Table 15 Key Functions for Arrow Keys with the **keybind** Command

Arrow Key	Default Key Binding	Key Function
Up arrow	Esc-[A	**previous_history**
Down arrow	Esc-[B	**next_history**
Left arrow	Esc-[C	**backward_char**
Right arrow	Esc-[D	**forward_char**

Restrictions

It is not possible to rebind the Tab, Space, Meta-Tab, or Meta-Space keys.

See Also

alias

language selection

customizing the environment according to language

You can customize the ObjectCenter programming environment according to language as follows:

- You can configure ObjectCenter to come up as either a C or C++ programming environment.
- If you use C in the environment, you can use different configurations, according to which of the following you wish to emulate: **suncc**, **acc**, **gcc**, **hpcc**, **hpc89**, or **clcc**.
- You can change the language configuration from one module to another so that, for instance, you can mix **cc**, **clcc**, and **C++** in one application.
- You can specify the language mode you want to use in the Workspace separately from the settings for the environment or the C compiler. For instance, you can use ObjectCenter as a C++ development environment, but use the Workspace in C mode from time to time when you need to look at lower-level details.

Choosing C or C++ as your programming environment

By default, ObjectCenter starts up as a C++ programming environment. To have the product come up as a C programming environment instead, put the following in your startup script:

```
setopt primary_language C
```

Using different C compiler configurations

To specify the default C language configuration, use the **config_c_parser** command. This command is effective only in C mode; it affects the C to be used when you type code in the Workspace or load code into ObjectCenter. The **config_c_parser** command has no effect on C++, nor does it have any effect on the setting of the **primary_language** option.

See the **config_c_parser** entry on page 103 for more information about this command.

Using module-specific configurations

To specify the language for interpreting a particular module, you can use the switches and options in Table 16; they are listed in descending order of precedence.

Table 16 Precedence of Switches and Options for Language Selection

Switches and Options	Description
-C and **-CXX** switches	To force a source or object file to be loaded as C, use **-C**. To force a source or object file to be loaded as C++, use **-CXX**.
c_suffixes and **cxx_suffixes** options	To force a source file to be loaded as C, use a suffix on the list specified by the **c_suffixes** option. To force a file to be loaded as C++, use a suffix on the list specified by the **cxx_suffixes** option. If the default language is C, the **c_suffixes** list takes precedence over the **cxx_suffixes** list. If the default language is C++, the **cxx_suffixes** list takes precedence over the **c_suffixes** list. By default, the **cxx_suffixes** list contains **.C** and **.c** suffixes and the **c_suffixes** list contains the **.c** suffix.
primary_language option	If you do not specify the language for a particular module with its suffix or with the **-C** and **-CXX** switches, then ObjectCenter uses the language specified by **primary_language**.

In other words, suppose the current default language is C++. Then ObjectCenter determines the source file language according to the following order:

1. If you specify **-C** and / or **-CXX**, these switches unconditionally determine source language.
2. If you do not specify **-C** and / or **-CXX**, ObjectCenter examines the **cxx_suffixes** list. If the filename suffix is listed there, then C++ is used.

3 If a filename's suffix is not on the **cxx_suffixes** list, ObjectCenter examines the **c_suffixes** list. If the filename suffix is listed there, then C is used.

4 If a filename's suffix is not on the **c_suffixes** list, ObjectCenter uses C++, the default language.

Similarly, suppose the current default language is C. Then ObjectCenter determines the source file language according to the following order:

1 If you specify **-C** and / or **-CXX**, they unconditionally determine the source language.

2 If you do not specify **-C** and / or **-CXX**, ObjectCenter examines the **c_suffixes** list. If the filename suffix is listed, then C is used.

3 If a filename's suffix is not on the **c_suffixes** list, ObjectCenter examines the **cxx_suffixes** list. If the filename suffix is listed, then C++ is used.

4 If a filename's suffix is not on the **cxx_suffixes** list, ObjectCenter uses C, the default language. The particular variant of C is selected according to the **config_c_parser** command.

Loading object files

If you do not specify **-C** or **-CXX** when you load object files, ObjectCenter uses the following rules:

- If the primary language is C, then ObjectCenter uses C.
- If the primary language is C++, ObjectCenter figures out whether to use C or C++ based on certain characteristics of the object code, primarily name-mangling.

Specifying the language mode for the Workspace

By default, the Workspace language mode is the same as the current primary language setting. You can change it with the **cmode** and **cxxmode** commands. Using **cmode** and **cxxmode** to change the language used in the Workspace has no effect on the language ObjectCenter uses to load source files.

link

links files from libraries and invokes automatic template instantiation if necessary

cdm	pdm
✔	

Command syntax

link

link -list

link *function*

link *variable*

Description

<< *none* >>	Attempts to satisfy references to all undefined variables and functions, including templates.
function	Resolves the undefined variables and functions used by the specified function.
variable	Resolves the undefined variables and functions used as initialization values for the variable.

Switches

-list	Echos the library link order to the Workspace. This switch is useful for diagnosing link-order related problems in the interpreter. The **link** command makes no links when used with the **-list** switch.

Usage

Use the **link** command to search all attached libraries to satisfy references to undefined variables and functions including templates. When you issue the **run** command, ObjectCenter automatically invokes its linking process, if necessary.

NOTE You may have to use the **link** command several times to eliminate all unresolved references.

By default, ObjectCenter displays a message when linking from a library:

```
Linking from ... Linking completed.
```

You can suppress the linking messages by setting the environment variable **CENTERLINE_LINK_SILENT** before starting ObjectCenter. This is particularly useful in Ascii ObjectCenter when linking from shared libraries: run-time linking messages will not obscure your program's output.

The **link** command automatically invokes the template instantiation system if your program uses any templates, so you might see loading and linking messages from the template instantiation system. Also, you might get syntax errors during the final linking phase of instantiation, since templates are instantiated later than the rest of your program is loaded and linked. If you do get syntax errors at the instantiation phase, edit only the template definition file, not your application file.

See the **templates** entry on page 335 for more information about the template instantiation system.

See Also **load**, **templates**, **unload**, **unres**, **xref**

list

displays source code lines

cdm	pdm
✔	✔

Command syntax

list
list *file*
list *"file":line*
list *function*
list *identifier*
list *line_number*
list *-number*
list *start_line end_line*

Description

<< *none* >>	Lists source code starting at the current list location.
file	Lists source code starting at the top of the specified file.
"file":line	Lists source code starting at the specified line number in the specified file.
function	Lists source code starting at the top of the specified function.
identifier	Lists source code starting at the line where the definition of the identifier (variable, typedef,macro, or class / struct / union tag) begins.
line_number	Lists source code starting at the line number specified.

-number	(The *number* argument preceded by a minus sign.) Lists source code starting at the specified number of lines *before* the current list location.
start_line *end_line*	Lists source code starting at the line number specified by *start_line* and ending at the line number specified by *end_line*.

Options

The following ObjectCenter options affect the **list** command:

cxx_suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.
list_action	(Ascii ObjectCenter only) Displays actions that execute everywhere when listing the source lin e at which they were triggered.
page_list	(Ascii ObjectCenter only) Sets the number of lines of source code the **list** command displays before a **more** prompt is issued.
path	Specifies the search path for listing files.
tab_stop	Specifies the number of spaces to indent per tab character when listing source code.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **list** command to display specific lines of source code relative to the current **list location**. The list location is set by the following events:

- When a file is loaded, it is set to the first line.
- When a break level is entered, it is set to the break location.
- When the **list** command is used, it is set to the last line displayed.

You can also set the list location using the **file** command.

In process debugging mode, if you use the **list** command and specify a static function for the *function* argument, you may receive an error in certain situations. However, if you first use the **whatis** command and specify the static function as an argument, the debugger loads additional symbols. Then, you can use the **list** command with the static function to show the source code in the Source area.

In Ascii mode, each time **list** is called, ObjectCenter displays **page_list** lines of source code. Both Motif and OPEN LOOK offer scrollbars to continue viewing more lines.

In component debugging mode (not in **pdm**), you can specify the location of a variable in one of four ways:

- *file`function`variable*
- *file`line_number`variable*
- *file`variable*
- *function`variable*

The more prompt responses for Ascii ObjectCenter listing

In Ascii ObjectCenter, the lines listed are followed by a "more" prompt that accepts the following responses:

h	Displays additional responses accepted
q	Quits the listing
Return	Shows one more line
Space	Displays another **page_list** lines of source code

In Ascii ObjectCenter, signals and errors are noted after the source line on which they occurred.

See Also **display**, **edit**, **load**, **whatis**, **whereis**

list_classes

lists the names of all classes loaded

cdm	pdm
✔	

Command syntax **list_classes**

Description

<< none >>	Motif and OPEN LOOK: Invokes the Class Hierarchy Browser. Ascii ObjectCenter: Lists the names of all classes currently loaded.

Usage Use **list_classes** to list the names of all the classes that are currently loaded.

See Also **browse_base**, **browse_class**, **browse_data_members**, **browse_derived**, **browse_friends**, **browse_member_functions**, **classinfo**

listi

displays machine instructions

cdm	pdm
	✔

Command syntax

listi
listi *addr*
listi *addr1 addr2*
listi *line*
listi *line1 line2*
listi *func*
listi *func + offset*

Description

<<*none*>>	Displays machine instructions at current program counter address.
addr	Displays machine instructions at *addr*. The value of *addr* can be a hexadecimal or octal number.
addr1 addr2	Displays machine instructions between *addr1* and *addr2*. The values of *addr1* and *addr2* can be hexadecimal or octal numbers.
line	Displays machine instructions at *line* in current file. The value of *line* must be a decimal number.
line1 line2	Displays machine instructions between *line1* and *line2*. The values of *line1* and *line2* must be decimal numbers.
func	Displays machine instructions for *func*.
func + offset	Displays machine instruction at the address equal to the address of *func* plus *offset*.

See Also

list, **nexti**, **stepi**, **stopi**

load

loads source, object, library, and project files

cdm	pdm
✔	

NOTE To load an **a.out** file, use the **debug** command in **pdm**. See the **debug** reference page for more information.

Command syntax

load [*switches* **]** *file* **...**

Description

[*switches* **]** *file* ... Loads specified files into ObjectCenter. If the specified files are already loaded, reloads files that have been modified since they were last loaded.

Files can be source, object, library, and project files, or template instantiation modules; see 'Files' on page 191 for more details.

Switches

The **load** command accepts all switches used with the C++ translator or the C compiler, but it acts upon only on the following switches:

+k[=*filename***]** When loading an object file that requires compilation outside the environment: save and restore header files from a repository (**+k**); if *filename* is provided, use it to determine which header files to skip. By default, header files are not saved and restored from the repository. See the **precompiled header files** entry on page 271 for more information.

-C	Cause ObjectCenter to parse the specified file as C code, rather than C++ code. See the **language selection** entry on page 175 for details about configuring the language used by ObjectCenter.
-CXX	Cause ObjectCenter to parse the specified file as C++ code, rather than C code. See the **language selection** entry on page 175 for details about configuring the language used by ObjectCenter.
-dd=off **-dd=on** (the default)	Specify whether ObjectCenter uses demand-driven code generation; see the **demand-driven code generation** entry on page 129 for more information. The default setting is **-dd=on** except for header files.
-D*name***[=***definition*]	Define *name* as if with a **#define** directive. If *definition* is not supplied, then define *name* as **1**.
-G	When loading compiled files, ignore debugging information produced by the **-g** switch of the compiler. This allows you to load compiled files for which ObjectCenter has trouble reading the debugging information. Also, you can save memory by loading libraries that have been debugged with **-G**; if you use **-G** when loading a library, ObjectCenter ignores debugging information when linking from the library.
-hdrepos=*directory*	When loading an object file that requires compilation outside the environment, look in *directory* for the *filename* (precompiled header information file) used with **+k***filename*. See the **precompiled header files** entry on page 271 for more information.

-I*directory_name*	Add *directory_name* to the list of directories to search for files specified by the **#include** preprocessor directive. When the name of a file is surrounded by double quotes (" "), the search path is as follows: first, the directory of the file being read, then in directories specified by **-I**, and finally in the **/usr/include** directory. When a filename is surrounded by angle brackets (< >), the search path is as follows: first in directories specified by **-I** and then in the **/usr/include** directory.
-L*dir*	Add *dir* to the list of directories to search for libraries.
-l*x*	Search for and load a library named libx.a, where *x* is a library name suffix. If shared libraries are supported by ObjectCenter on your platform, see the *ObjectCenter Platform Guide* for information about loading them.
-U*macro_name*	Cause the predefined *macro_name* to become undefined as if by an #**undef** directive.
-w	Suppress warnings, but report errors. If you use **-w** when loading a library, warnings are suppressed when modules are linked from the library.

Options

The following ObjectCenter options affect the **load** command.

NOTE Whenever you issue the **load** command with a particular file, ObjectCenter uses the option values that were in effect the first time the file was loaded. If you change the value of an option after loading a file, and you want that option to affect the file, you must explicitly issue an **unload** command for the file and then reload it. This is true even if the file failed to load when you issued the **load** command.

---

ansi	Performs preprocessing in strict conformance with the ANSI C Standard.
auto_compile	Automatically compiles missing or outdated object files.
batch_load	(Ascii ObjectCenter only) Suppresses prompts to the user during loading.
cc_prog	Specifies the name of the C compiler that ObjectCenter invokes.
ccargs	Specifies arguments passed to the C compiler when invoked from ObjectCenter.
create_file	Specifies commands to create a new file when loading.
cxx_prog	Specifies the name of the C++ translator to be invoked by ObjectCenter when compiling a C++ file.
cxx_suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.
cxxargs	Specifies default arguments passed to the C++ translator when invoked by ObjectCenter.
echo	Echoes the input stream after preprocessing (similar to the **-E** compiler switch).
ignore_sharp_lines	Causes ObjectCenter to ignore #**line** directives generated by preprocessors.
instrument_all	Automatically instruments files as they are loaded. See the **instrument** entry on page 162 for more information.
lint_load	Indicates the severity of warnings issued when loading files, or suppresses warnings if set to **0**.
load_flags	Specifies the default switches to use if **load** is called without any switches. ObjectCenter always uses any **-L** switches specified in **load_flags**.

long_not_int	Specifies whether **long** and **int** are treated as the same type.
page_load	(Ascii ObjectCenter only) Sets the number of lines of error reports to display before prompting the user for more.
path	Specifies the search path for loading source and object files (not for **#include** files).
preprocessor	Specifies a command to execute in a subshell before the file is loaded.
proto_path	Specifies search path for prototype files.
src_err	(Ascii ObjectCenter only) Specifies the number of source lines to be listed for errors and warnings.
subshell	Specifies the shell used to invoke the C++ translator or C compiler.
sys_load_cflags	Specifies switches that establish the search path for system libraries and **#include** files when loading C source files with **-C**.
sys_load_cxxflags	Specifies the switches that establish the search path for system libraries and **#include** files when loading a C++ file.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **load** command to load files into ObjectCenter or reload files that have been modified since they were loaded.

Using system-wide loading switches

When loading files, ObjectCenter always uses command-line switches specified by one of two options: either **sys_load_cflags** for (C files) or **sys_load_cxxflags** (for C++ files). The **sys_load_cflags** and **sys_load_cxxflags** options specifythe directories to search for libraries and system header files as well as some macros.

ObjectCenter's default values for **sys_load_cflags** and **sys_load_cxxflags** are specified in the system-wide **ocenterinit** file. The exact values depend on the type of workstation you are using. To see the values on your system, enter this command:

```
-> printopt sys_load_cxxflags
```

and

```
-> printopt sys_load_cflags
```

If you have a different library or **#include** path for either **CC** or **cc** from that specified by the **sys_load_cxxflags** or **sys_load_cflags** option, you should change the value of the option either in your personal **.ocenterinit** file or in the system-wide **ocenterinit** file.

Specifying your own loading switches

ObjectCenter always uses all switches specified with **sys_load_cxxflags** and **sys_load_cflags**. In addition, ObjectCenter also uses any switches specified with the **load_flags** option.

Typically, you use the **load_flags** option to specify any switches specific to your own work. For example, the following commands show the use of a macro name, **BETA**, specific to a project:

```
-> setopt load_flags -DBETA
-> load xyz.c
Loading(C++): -DBETA xyz.c
->
```

With one exception, any switches you explicitly enter when you issue **load** replace all switches you may have specified with **load_flags**. The exception is the **-L** switch in **load_flags**; ObjectCenter always uses **-L** switches specified by **load_flags**.

Here is an example:

```
-> setopt load_flags -DBETA -w
-> load sample.c
Loading: -DBETA -w sample.c
-> unload sample.c
Unloading: sample.c
-> load sample.c -DDEBUG
Loading: -DDEBUG sample.c
```

In this example, when we explicitly specify the **-DDEBUG** switch when loading the file **sample.c**, ObjectCenter uses **-DDEBUG** instead of the **-DBETA** and **-w** switches specified by **load_flags**.

TIP: When does the load_flags option have precedence?

If you are loading a file for the first time, and if you do not specify any switches with the **load** command, ObjectCenter uses the switches specified by the **load_flags** option.

However, if you are loading a file for the first time and you do specify any switches with the **load** command, ObjectCenter uses the switches you specify with **load** instead of the switches in the **load_flags** option.

After the first time you load a file, ObjectCenter reuses the switches it used the first time it loaded the file whenever it attempts to load that file. For instance, when you reload a file by issuing **load** in the Workspace without any switches, ObjectCenter reuses the switches from the first time you loaded the file. Similarly, if you reload the file by issuing a **build** command, ObjectCenter reuses the switches from the first time it loaded the file.

Once you have loaded a file, changing the value of **load_flags** has no effect on subsequent loads of that file, even if the **load_flags** option was applied the first time you loaded it.

If you want to change the switches that ObjectCenter uses when loading a file that has already been loaded, you must do one of the following:

- Issue the **load** command in the Workspace using the new switches. Then ObjectCenter will use the new switches every time it attempts to load the file.
- Change the value of **load_flags** to specify the new switches, issue an **unload** command for the file, and then issue a **load** command for the file without specifying any switches. In this case, ObjectCenter uses the switches specified in **load_flags.**
- Use the file's property sheet in the Project Browser to change the options used to load the file.

Files

Use the **load** command to load the following kinds of files:

- Source, including preprocessed source files and **#include** files
- Object
- Library files, including prototype files
- Project
- Template instantiation modules

We describe loading each kind of file in the next few sections. See the **performance** entry on page 263 for an overview and the **debugging** entry on page 116 for a more detailed discussion of trade-offs in the way you load your program.

Loading source files

If you issue **load** with the name of a source code file when the corresponding object code file is already loaded, ObjectCenter unloads the object file before it loads the source file.

Loading C++ source files

By default, source files are loaded as C++ modules. C++ source files are loaded and translated into an intermediate code that is used when you execute the code.

The translation process is compatible with the C++ translator and Release 3.0 of the AT&T C++ Language System defined in the *AT&T C++ Language System Release 3.0 Product Reference Manual* supplied with ObjectCenter. For information about C++, see that manual.

For more information about ObjectCenter's compatibility with other C++ translators and with Release 3.0 of the AT&T C++ Language System, see the .

Using the cxx_suffixes option to specify extensions

The **cxx_suffixes** option specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file. When loading files, this condition arises if you issue the **load** command and specify an object file. In this case, ObjectCenter needs to check the corresponding source file to see if the object file is up-to-date. If not, ObjectCenter compiles a new object file before proceeding.

By default, the **cxx_suffixes** option contains the **.C** and **.c** suffixes. If you are in C++ mode, which is also the default, this setting for **cxx_suffixes** means that ObjectCenter first searches for a corresponding C++ source file using a **.C** extension. If the search with this extension fails, then ObjectCenter searches using a **.c** extension.

If you have specified other, or additional, file extensions with the **cxx_suffixes** option, ObjectCenter first searches for a corresponding C++ source file using the extensions in the order they are specified, left to right. If searches with all specified extensions fail, then ObjectCenter searches using a **.C** extension and then a **.c** extension.

For example, if you set the **cxx_suffixes** option as follows:

```
-> setopt cxx_suffixes cxx cpp
```

and then issue the following command:

```
-> load bar.o
```

ObjectCenter searches for a C++ source file corresponding to **bar.o** in the following order: **bar.cxx**, **bar.cpp**, **bar.C**, and **bar.c**.

NOTE See the **alias** entry on page 174 for more information about setting the language ObjectCenter uses.

Loading C source files

To load a source file as a C module, use the **-C** switch.By default, ObjectCenter loads C source files following the same rules that the **cc** command follows on your system. The source files are translated into an intermediate code that is used when you execute the code. The translation process is compatible with the C compiler and the ANSI standard for the C language.

Use the **config_c_parser** command to change the compiler configuration used by ObjectCenter, and use the **ansi** option to check for compliance with the ANSI standardmake sure that the C preprocessor is ANSI compliant. See the **cc and other C compilers** entry on page 40 and the **ANSI C** entry on page 12, respectively, for additional information about compatibility between ObjectCenter, C compilers, and the ANSI C standard.

Loading and using preprocessor files

ObjectCenter uses **#line** directives to map certain kinds of preprocessed code to the unpreprocessed code. It therefore allows you to work directly with input files that are run through preprocessors that generate C or C++ files with **#line** directives pointing back to the input file. Such preprocessors include **yacc** and certain SQL preprocessors. ObjectCenter uses the **#line** directives to associate lines in the generated C or C++ file with lines in the input file that you wrote.

Thus, ObjectCenter helps you debug preprocessed code by allowing you to examine the input to a preprocessor rather than just the output from it; the input is typically much easier to read than the output. To work with preprocessor files:

1. Load a file containing **#line** directives.
2. Work with the input file in your ObjectCenter session.

NOTE See the **preprocessed code** entry on page 276 for more information about debugging code generated by preprocessors such as **yacc**.

Search path for #include files

To give the search path for **#include** directories, use the **-I** switch with the **load** command according to the following format:

load -I*include_dir1* **[-I***include_dir2* **...]** *file*

NOTE The **path** option does not provide a search path for loading **#include** files, only for loading source and object files.

Loading object files

You can load object code files that have been compiled with or without the **-g** compiler switch that adds debugging information. However, to have the greatest debugging functionality in ObjectCenter, load object code files compiled with debugging information whenever possible.

NOTE When loading object code into ObjectCenter, make sure that the object code was compiled with the same release of the operating system that you are using to run ObjectCenter.

Because object files do not retain information about classes, ObjectCenter does not provide full source-level debugging, class browsing, and Workspace interaction for classes defined in C++ code loaded in object form. See the "Special considerations with C++ object files" section on page 123 for more information.

To load an object file as a C module, use the **-C** switch.

If you issue **load** with the name of an object code file when the corresponding source code file is already loaded, ObjectCenter unloads the source file before it loads the object file.

If an object code file specified with **load** does not exist and the directory that contains the source file contains a makefile, ObjectCenter does a **make** of the object file. Otherwise, if the source is available, ObjectCenter creates an object file by calling the C++ translator or the C compiler.

ObjectCenter supports the loading of CenterLine-C object files as well as those generated by your platform's native C compiler. See the *ObjectCenter Platform Guide* for your particular platform for information about any additional object files that ObjectCenter may support.

Specifying a different compiler

When ObjectCenter needs to compile a C file, it invokes the compiler defined by its **cc_prog** option. If this option is unset (the default), ObjectCenter invokes **cc**.

If ObjectCenter can't find the file

If you specify a file with **load** that does not exist, ObjectCenter looks at the setting of its **create_file** option. If **create_file** describes how to create the specified file, ObjectCenter uses those instructions to create the file, then loads it. For example:

```
-> ls *.C
backup.C
-> load a.C
Cannot open '/net/fenway/u1/bobh/code/a.C'.
-> setopt create_file @a.C@cp backup.C a.C
-> load a.C
Cannot open '/net/fenway/u1/bobh/code/a.C'.
Executing: cp backup.C a.C
Loading (C++): a.C
```

For more information about **create_file**, see the **options** entry on page 229.

Loading libraries

Loading a library makes the contents of the library available to ObjectCenter. You can load a library by:

- Specifying the full pathname of the library with the **load** command
- Using the **-l** switch.

  This is similar to using the **-l** switch to **cc**. See Table 17 for a listing of the order in which ObjectCenter searches directories for the library.

Table 17 ObjectCenter's Search Path for Libraries

Order	Search Path
1	Directories specified on the command line by **-L***dir* in the order specified
2	Directories specified by **-L** in ObjectCenter's **load_flags** option
3[a]	Default system directory specified by the **sys_load_cxxflags** and **sys_load_cflags** option

a. See the "Specifying the search path for loading libraries and #include files" **TIP** on page 196.

TIP: Specifying the search path for loading libraries and #include files

If you are using an ANSI compiler, make sure that the path for the libraries and **#include** files required for ANSI are specified either on the **load** command line or by setting the **load_flags** and / or **sys_load_cflags** options. The directories required by ANSI must be searched *before* the default system directory specified by **sys_load_cflags.** You must also explicitly load the C library.

Similarly, if you use a compiler like **clcc** that has header files and libraries in "non-standard" locations, be sure to set the switches to the **load** command to specify the correct location to search before the default system directory specified by **sys_load_cflags**. You must also explicitly load the C library.

For instance, if you are using **clcc** as your C compiler and using **-ansi** as a compilation mode, you should make the following specifications:

- Issue the **setopt ansi** command.
- Set the **sys_load_cflags** option to contain the following as the first **-L** specification:

  -L/usr/local/CenterLine/clcc/*arch_os* / **lib**

  where *arch_os* is the name of your architecture and operating system.
- Set the **sys_load_cflags** option to contain the following **-I** specification before the specification of **-I/usr/include**:

  -I/usr/local/CenterLine/clcc/*arch_os***/inc**

  where *arch_os* is the name of your architecture and operating system.
- Issue the following command:

```
->load /usr/local/CenterLine/clcc/arch_os/lib/libc.a
```

NOTE If shared libraries are supported by ObjectCenter on your platform, see the *ObjectCenter Platform Guide* for information about loading them.

Some operating systems provide a **-u** *symname* option to **ld**, which allows you to enter *symname* as an undefined symbol in the symbol table. The **-u** option is typically used to load entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

ObjectCenter does not provide a **-u** switch for the **load** command. Nonetheless, you can force the loading of a first function from a library in ObjectCenter by defining the function as external and making a reference to it. Here is an example:

```
1 -> extern void main ();
2 -> main;
```

NOTE For some of the C library functions, you can substitute your own version. See your *ObjectCenter Platform Guide* for a list of the C library functions replaced by ObjectCenter, and the ones that you can replace.

To use your own version of a function, load the function in a source or object file before linking your program. If your program has already been linked, you must quit, then start a new ObjectCenter session to substitute your function for one of the ObjectCenter replacements.

Loading function prototype files

When working with C files, loading function prototypes allows ObjectCenter to check the number and type of arguments for calls to functions. Prototype files conventionally end in **.proto**. If a filename ends with **.proto**, **load** first looks for the file in the current directory, then looks in the list of directories specified by the **proto_path** option.

For information about creating your own prototype files, see the reference page for the **proto** command.

Loading project files

If the first line of the file you specify with **load** is as follows:

```
/* ObjectCenter Project File */
```

ObjectCenter will invoke **source** instead of **load** to retrieve the file's contents. This is the way in which ObjectCenter loads project files. When you load a project file, ObjectCenter reloads the most recent version of the source and object files in your project.

NOTE Loading a project file does *not* unload any modules that were already loaded. To unload modules before loading a project file, use the **unload** command first.

Using shell wildcards

The **load** command takes shell wildcards so you can load groups of files with one command. For example:

```
-> load str*.C
Loading (C++): str_1.C
Loading (C++): str_2.C
Loading (C++): str_3.C
```

Using wildcard expansion

If you use shell wildcards with **load**, you can also use **Esc-x** at the end of a command line to expand these wildcards. The escape sequence echoes the command line to a subshell that expands any wildcards. Here's an example:

```
-> load *.C f?.o<Esc-x>
-> load abc.C xyz.C f1.o f2.o<Return>
Loading C++: abc.C
Loading C++: xyz.C
Loading C++: f1.o
Loading C++: f2.o
->
```

ObjectCenter pauses after displaying the expanded command line, allowing you to edit the command line before executing it.

The sequence **Esc-x** is one of the key bindings supported by the Workspace. See the **keybind** entry on page 167 for more information.

Disabling load-time error checking with comments

You can suppress certain kinds of error checking by using predefined comments in your source code. See the **built-in comments** entry on page 33 for more information.

Restrictions

Loading an object file without debugging information may cause spurious warnings since initialized variables can be grouped together without correct type or size information.

If **load** rejects an object file that was compiled with debugging information (for example, due to a type redeclaration), try loading the file with the **-G** switch.

Occasionally, linking libraries may produce spurious warnings about size or type redeclarations.

Trying to reload a file in the Workspace by using **load** with different switches does not necessarily cause ObjectCenter to reload the file. Unless the file itself has been modified, ObjectCenter considers it up-to-date and will not reload it. You can work around the problem by using the **unload** command and then **load** with the desired switches.

See Also

built-in macros, **config_c_parser**, **contents**, **debugging**, **make**, **save**, **swap**, **unload**

load_header

loads header files as source

cdm	pdm
✔	

Command syntax

load_header [*switches* **] {***file.h* **... | <***file.h***>... |** "*file.h*"**...}**

Description

[*switches* **]** *file.h* ...
[*switches* **]** "*file.h*" ...
Loads specified header files into ObjectCenter, searching in the current directory first, and then in the directories specified below.

[*switches* **] <***file.h***>** ...
Loads specified header files into ObjectCenter, searching in the directories specified below.

If there are any switches specified on the **load_header** line, ObjectCenter searches for header files in the directories specified with **-I** on the **load_header** line, if any, then in the directories specified in the **sys_load_flags** (for C files) or **sys_load_cxxflags** (for C++ files) options.

If there are no switches specified on the **load_header** line, ObjectCenter searches for header files in the directories specified in the **load_flags** option, then in the directories specified in the **sys_load_cflags** or **sys_load_cxxflags** options. ObjectCenter loads the header files with demand-driven code generation turned off, unless you use the **-dd=on** option on the **load_header** command line or in your **load_flags** option.

Switches

The **load_header** command accepts all the switches that the **load** command accepts. It ignores switches that have no meaning in the context of loading header files.

Options

The **load_header** command is affected by the same options that affect the **load** command. Please refer to the **load** entry for details.

NOTE Whenever you issue the **load_header** command with a particular file, ObjectCenter uses the option values that were in effect the first time the file was loaded. If you change the value of an option after loading a file, and you want that option to affect the file, you must explicitly issue an **unload** command for the file and then reload it. This is true even if the file failed to load when you issued the **load_header** command.

Usage

Use the **load_header** command to load the definitions from one or more header files without specifying a path, or to load definitions from multiple header files into a single module. The definitions are loaded into the environment in a separate file. The **load_header** command replaces the **#include** syntax used in previous releases of ObjectCenter. See 'Compatibility with previous releases' on page 203 for more information.

NOTE If the header file you wish to load is in your working directory or path, you can use the **load** command to load it.

The first time you use the **load_header** command in an ObjectCenter session, ObjectCenter creates a directory called **OC.***pid* in the **/tmp** directory, where *pid* is a process id, and creates a file in that directory. For the rest of the ObjectCenter session, **load_header** uses the same **OC.***pid* directory.

If you specify only one header file on the command line, ObjectCenter creates a file with the name of the included file. The file contains a single **#include** directive. For example:

```
C++ -> load_header <iostream.h>
Loading (C++): -I./tmp/OC.18be/iostream.h
C++ -> sh more /tmp/OC.18be/iostream.h
#include <iostream.h>
```

You can load multiple header files into a single module. You will want to do this if a header file has dependencies on definitions in other header files. In this case, ObjectCenter names the module **_load_header_files__*n*.h**, where *n* is a unique hexadecimal number. For example:

```
C++ -> load_header <math.h> <limits.h> "rect.h"
Loading (C++):-I./tmp/OC.18be/_load_header_files__1.h
C++ -> sh more /tmp/OC.18be/_load_header_files__1.h
#include <math.h>
#include <limits.h>
#include "rect.h"
```

You can use the Project Browser or the **contents** command to examine the contents of the modules. The **contents -ascii** command lists all loaded files, and **contents -ascii** with the name of the module lists all the symbols defined in the file. For example:

```
C++ -> contents -ascii
object: centerline (C++)
source: workspace (C++)
library: /lib/milli.a
library: /lib/libc.sl
library: /tmp_mnt/net/plough/u5/demos/codecenter/pa-hpux8/lib/libC.a
source: /tmp/OC.18be/_load_header_files__1.h (-I.)
C++ -> contents -ascii /tmp/OC.18be/_load_header_files__1.h
Contents of   source: /tmp/OC.18be/_load_header_files__1.h (-I.) (C++)
/usr/include/math.h
/usr/include/sys/stdsyms.h
/usr/include/limits.h
/usr/include/sys/param.h
...
typedef long fd_mask ;
struct fd_set {...} ;
typedef struct fd_set fd_set ;
struct tm {...} ;
struct timeval {...} ;
struct timezone {...} ;
struct itimerval {...} ;
struct ki_timeval {...} ;
struct entry {...} ;
int number ;
struct entry *list_head ;
C++ ->
```

There are several ways to unload a header file module:

- Highlight its name in the Project Browser and select the Unload button
- Use the **unload** command in the Workspace with the full pathname of the file
- If the module contains a single **#include** directive, use the **unload** command with the filename, for example:

```
C++ -> unload iostream.h
Unloading: iostream.h
```

Compatibility with previous releases

In previous releases of ObjectCenter, you could load a header file into the Workspace by using a **#include** directive in the Workspace. This sometimes caused confusing problems.

For example, because definitions were parsed one at a time, if an error was encountered while parsing a header file, previous definitions were not undefined. As a result, users often had to issue an **unload workspace** command before they could reload a header file.

The Workspace does not match the separate compilation model of C and C++; to enable the Workspace to function as a debugger, definitions in a header file included in the Workspace are visible across modules. As a result, using **#include** in the Workspace occasionally caused ObjectCenter to pick up incorrect definitions from included files.

For users with existing project files that use the **#include** syntax, we have introduced a new option, **workspace_include**. When this option is set, you can use **#include** in the Workspace. We recommend that you add this line to the beginning of any project file that uses **#include**:

```
setopt workspace_include
```

and this line to the end of the project file:

```
unsetopt workspace_include
```

See Also

built-in macros, **config_c_parser**, **contents**, **debugging**, **load**, **make**, **save**, **swap**, **unload**

make

invokes the UNIX **make** command to handle CenterLine (CL) targets

cdm	pdm
✔	✔

Command syntax

make

make *target ...*

Description

<< *none* >>	Calls the UNIX **make** command using the default target. Motif and OPEN LOOK: Shows load-time errors in the Error Browser.
target ...	Calls the UNIX **make** command using the *target* argument as its target. Motif and OPEN LOOK: Shows loadtime errors in the Error Browser.

NOTE Using the **make** command while you are in process debugging mode has the same effect as using the **make** command in the shell; it does not recognize any CL target rules. The following description of the **make** command applies to component debugging mode only.

Options

The following ObjectCenter options affect the **make** command:

cc_prog	Specifies the name of the C compiler that ObjectCenter invokes.
ccargs	Specifies arguments passed to **cc** when invoked from ObjectCenter.

cxx_prog	Specifies the name of the C++ translator to be invoked by ObjectCenter when compiling a C++ file.
cxxargs	Specifies default arguments passed to the C++ translator when invoked by ObjectCenter.
make_args	Specifies the command-line arguments passed to the UNIX **make** command by ObjectCenter's **make** command.
make_hfiles	Checks header files to find out if a file should be reloaded.
make_offset	Specifies the number of characters to skip when reading shell commands from the **make** program.
make_prog	Specifies the program invoked to make a target (the default is **make**).
make_symbol	Specifies the string used to denote an ObjectCenter command line in a makefile. By default, the string is the **#** character.
subshell	Specifies the shell used to invoke the C++ translator or C compiler.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the ObjectCenter **make** command to load files into ObjectCenter using makefiles containing CL target rules in addition to the standard target rules. See the UNIX manual page for **make** for a list of the switches you can use.

ObjectCenter's EZSTART utility provides a shortcut for creating CL targets in makefiles; see the **clezstart** entry on page 72 for more information.

What is a CL target rule?

A standard UNIX **make** target rule contains **shell lines**, which are lines containing shell commands. The syntax for a shell line is as follows:

<tab>*shell command***[;** *shell command* **...]**

For example, the following is a shell line in a standard UNIX makefile target:

```
<tab>echo “starting a standard target”
```

A *CL target rule* is just like a standard target rule except that it contains one or more CL lines; a *CL line* is a makefile command line preceded by **<tab>#**. ObjectCenter handles CL lines as ObjectCenter commands. It passes all other lines to the Bourne shell for execution, just as when **make** is used outside of ObjectCenter.

The syntax for a CL line in a CL target rule is as follows:

<tab>#*ObjectCenter command*

For example, the following is a CL line in a CL target rule:

```
<tab>#load a.o b.o
```

The preceding example has the same effect as the following command issued in the Workspace:

```
-> load a.o b.o
```

NOTE A **#** character in the first column in a line causes ObjectCenter to treat that line as a comment, so you must indent the # to indicate that an ObjectCenter command follows. Use the Tab key to indent.

Here is another example of a standard target and the corresponding CL target:

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

Designing a CL target

To design a CL target that you can add to a makefile, think of the ObjectCenter commands that you want your makefile to automate. For example, if your standard target is the following:

```
prog: a.o b.o
     echo "starting a standard target"
     $(CC) $(CFLAGS) -o my_program a.o b.o -lm
```

then the equivalent CL target would be the following:

```
cl_obj: a.o b.o
     echo "starting a cl target"
     #load $(CFLAGS) a.o b.o -lm
     #link
     #setopt program_name my_program
```

Example

NOTE See the **templates** entry on page 335 for more information about templates and how to use **make** with template object modules.

The following is an excerpt from a typical makefile that includes two standard targets used directly by **ccCC** (**.C.o** and **all**) and two that are specific to ObjectCenter (**ocenter_src** and **ocenter_obj**):

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

SRCS = a.C b.C c.C
OBJS = a.o b.o c.o
FLAGS = -g -DDEBUG
.SUFFIXES: .C .o
# The following is an implicit target that specifies
# how to convert a .C file to a .o file. In this case
# CC is called with the switches +d, -g, and -c
.C.o:
  CC +d -g -c $<

# The next target creates an executable named all
# from the three files a.o, b.o, and c.o. If any of
# the .o files are missing or out of date, they will
# be compiled, using the implicit target .C.o

all: $(OBJS)
  CC +d -g -o all $(OBJS)

# targets specific to ObjectCenter ...
# note the indented # character

ocenter_src: $(SRCS)
  #load $(FLAGS) $(SRCS)

# the following loads object files into ObjectCenter,
# using the implicit target to convert .cC to .o

ocenter_obj: $(OBJS)
  #load $(FLAGS) $(OBJS)
```

If you issue the **make** command within ObjectCenter and provide the name of a standard target, ObjectCenter passes the command line to the Bourne shell for execution:

```
-> make all
sh CC +d -g -c a.C
CC  +d a.C:
cc  -c  -g a.c
sh CC +d -g -c b.C
CC  +d b.C:
cc  -c  -g b.c
sh CC +d -g -c c.C
CC  +d c.C:
cc  -c  -g c.c
sh CC +d -g -o all a.o b.o c.o
cc  -L/usr/local/lib  -o all  -g a.o b.o c.o -lC
->
```

In this example, the **.o** files did not exist, so the **.C** files had to be compiled. After ObjectCenter passes the invocation of **CC** to the shell, **CC** processes the files, calling **cc** to compile and link them.

Target rules that call cc , CC,or ld

If an explicit target rule or an implicit suffix rule causes a call to **cc**, **CC**, or **ld**, the corresponding CL target rule should issue the **load** command on the same source or object files. Also, you need to supply the same switches with **#load** that you would use with **cc**, **CC**, or **ld** — for instance, the **-D** switch.

CL suffix rules for loading individual files

You can add implicit rules specific to ObjectCenter for loading individual files. For example, consider the following makefile fragment:

```
FLAGS = -g -DDEBUG
.SUFFIXES: .c .o .src .obj
.c.src:
  #load $(FLAGS) $<
.o.obj:
  #load $(FLAGS) $<
```

The first rule specifies that to make a file ending in .**src**, load a source file ending in **.c**. The second rule indicates that to make a file ending in **.obj**, load an object file ending with **.o**.

If you set up your makefile with these implicit rules, you can load individual source or object files by specifying a file with a **.src** or **.obj** suffix as a target:

```
-> make a.src
load -g -DDEBUG a.c
Loading: -DDEBUG a.c
-> make b.obj
load -g -DDEBUG b.o
Loading: b.o
```

Meta-character

Before ObjectCenter executes rules that begin with a #, it passes them first through the Bourne shell, just as **make** does. The subshell interprets all meta-characters and sends the output back to ObjectCenter.

To avoid the delay when spawning the shell, or to avoid improper meta-character expansion by the Bourne shell, preface the command with two # characters. For example, the following rule uses **##** to prevent the Bourne shell from interpreting the left and right parentheses as meta-characters.

```
start:
   #load $(FLAGS) $(SOURCES)
   ##printf("All done\n");
```

Other characters

See Table 18 for a description of the meaning and usage of various characters in CL targets.

Table 18 Meaning of Special Characters in CL Targets

Character	Meaning and Usage
\ character (backslash)	On CL lines and shell lines in CL targets, use the backslash to escape **EOL** in the same way as you do for shell lines in standard targets. Note that a backslash does not escape a space character on a CL line for the **load** command.

@ character	Execute but do not echo the current line; this does not apply to **nmake**.
	
	Beginning a shell line in a CL target with an **@** character does not interfere with the ObjectCenter **make** command's implicit use of the **-n** option. For example:
	
	`any_cl-specific_target: $(X_OBJ)`
	`  echo "next line not echoed by UNIX make"`
	`  @$(CC) $(CFLAGS) -DX -o xcompile bounce.c\`
	`  $(XOBJ) $(XLIBS)`

\" characters (escaped quotation marks)	Use the double CL target symbol (**##**) to keep escaped quotation marks (\") from being stripped. For example: `##load -DTIME=\"three_bells\" new.c` As shown in the example above, even with escaped quotation marks, you cannot pass a space character on a CL line for the **load** command. For more information, see the entry for "space character" next in this table.

space character	On a CL line, a space character cannot be passed in an argument for the **load** command. For example, there is no exact CL line equivalent of the following standard shell line:
	
	`$(CC) -DTIME=\"three\ bells\" foo.c.`
	
	One workaround is to eliminate the space in the macro definition in the following way:
	
	`##load -DTIME=\"three_bells\" foo.c`
	
	This limitation does not, however, apply to passing a space character on CL lines with ObjectCenter commands other than **load**. For example, the following CL line is valid:
	
	`#setenv TIME three bells`

CL lines that change directories

If you are designing a CL target that changes the current directory, keep in mind that the ObjectCenter **cd** command does not affect subsequent commands in the CL target in the same way that it affects subsequent commands in a standard target.

In a standard target, since each rule line invokes a new subshell, a **cd** shell command affects only subsequent commands on the same line. For example, in the following standard target, the **CC** in the second line of the rule will be invoked from the **new_dir** directory, while the **pwd** in both the first and third lines will be issued in the parent directory of **new_dir**:

```
standard_subs:
     pwd
     cd new_dir; $(CC) -c $(CFLAGS) a.c
     pwd
```

Since each CL line can have only a single CL command, the CL target equivalent for this standard target is the following:

```
cl_subs:
     #pwd
     #cd new_dir
     #load $(CFLAGS) a.o
     #cd ..
     #pwd
```

The **cd new_dir** command sets the current directory for the Workspace until the Workspace directory is explicitly reset by a new **cd** command. Therefore, the **cd ..** command returns the Workspace to the original directory so that the first and second **pwd** commands display the same directory.

CL targets that invoke make

To invoke **make** from a CL target, use a shell line and implement the call using **$(MAKE) -$(MAKEFLAGS)**. The **MAKE** macro causes the **make** utility to be executed immediately so that each lower-level makefile unwinds in the correct order. The **MAKEFLAGS** macro ensures that the proper switches are passed down from ObjectCenter.

NOTE For recursive invocations of **make** in CL targets, invoke **make** only from shell lines. That is, avoid the following CL line constructions: **#make**, **##make**, **#$(MAKE)**, and **##$(MAKE)**. Using these CL line constructions to invoke **make** may cause incorrect recursion and will give unpredictable results.

If both the recursive call and the new target being generated are in the same directory, then designing these CL targets is straightforward. For example:

```
cl_recursive:
     $(MAKE) -$(MAKEFLAGS) CFLAGS=-DFOO stopper
stopper:
     #load $(CFLAGS) a.o
```

However, if the call and the target are in different directories, you first use a shell line that both changes the directory and invokes **make**. For example, where **cl_switch** is in the makefile in **/dir1** and **cl_sub2** is in the makefile in **/dir1/dir2**:

```
cl_switch:
     cd dir2; $(MAKE) -$(MAKEFLAGS) \
     DIR=dir2 cl_sub2
```

Also, in the CL target for the makefile in the lower-level directory (here **/dir1/dir2**), you need to keep the ObjectCenter Workspace synchronized with the current working directory of the shell from which the recursive **make** was invoked. Synchronize the Workspace by using a pair of CL lines that issue the **cd** command. For example, with the recursive call to **make** in the target **cl_switch** shown above, the target in the new directory would use **cd** commands in the following way:

```
cl_sub2:
     #cd $(DIR)
     #load $(CFLAGS) a.o
     #cd ..
```

Debugging CL targets

Debug a CL target by using the **-n** switch as an argument to the ObjectCenter **make** command issued on the CL target you are testing.

When the **-n** switch is used as an argument, the ObjectCenter **make** command echoes but does not execute the rule.

Using **make -n** to debug a CL target from the Workspace is similar to debugging a standard target from the shell using **-n** with the UNIX **make** command. After issuing **make -n** in the Workspace on a CL target, check the listing of commands displayed in the Workspace to see if this is exactly the series of commands you want executed by ObjectCenter.

Loading changes into ObjectCenter

Use the **make** command to load your files into ObjectCenter at the start of a session. If you make changes to files that are loaded, use the **build** command to check the dependencies and reload all files that are affected, with the following exceptions:

- If you make changes that affect only a few files, and you know which files these are, the fastest way to load the changes is with the **load** command.
- If you change a makefile in a way that affects any CL targets used to load files currently in your ObjectCenter session, use the **make** command.

Compatibility

This section describes differences between ObjectCenter's **make** and other implementations.

SHELL makefile variable

With the UNIX **make** command, the **SHELL** variable specifies which subshell is invoked by a standard target. The ObjectCenter **make** command ignores makefile **SHELL** variable definitions, such as

```
SHELL = /bin/csh
```

To have the ObjectCenter **make** command use a shell other than **/bin/sh**, set the **shell** option and redefine the ObjectCenter **sh** command in the following way:

```
-> setopt shell /bin/csh
-> rename centerline_sh centerline_binsh
'centerline_sh' renamed to 'centerline_binsh'
-> int centerline_sh(a) char *a; {
+> centerline_shell(a);
+> }
```

nmake compatibility

Using the ObjectCenter **make** command with the AT&T version of the UNIX **make** utility (**nmake**) requires the adaptation of both CL targets and the ObjectCenter environment in several ways. For a detailed discussion of how to implement these adaptations for **nmake**, contact CenterLine Software Technical Support and request the Support Note *Using AT&T nmake with CodeCenter and ObjectCenter.*

gmake compatibility

Unlike many other versions of the UNIX **make** utility, GNU **make** (**gmake**) filters out and ignores all lines starting with a **TAB#**. To use **gmake** with the ObjectCenter **make** command, set the **make_prog** option to **gmake** and set the **make_symbol** option to something other than the **#** character; for example, set **make_symbol** to the **!** character. Then construct CL lines using this alternative CL target symbol.

For example, assume that you make the following settings in the Workspace or in your startup file:

```
-> setopt make_prog gmake
-> setopt make_symbol !
```

Then the CL lines in your CL targets would need to look like the following:

```
a_cl_target:
     !load $(CFLAGS) new1.o new2.o
```

Errors

Error messages related to the ObjectCenter **make** command are displayed in the Error Browser or the Workspace.

Restrictions

The makefile option called **.SILENT** does not work well with ObjectCenter's **make**.

ObjectCenter does not support the use of CL target rules in **pdm**.

In most cases, you can use the same switches with ObjectCenter's **make** command that you would use with the command outside the environment, with the following exceptions:

- The **-D** switch is *not* compatible with the ObjectCenter **make** command. Do not use **make -D** in the Workspace.
- The **-s** switch is *not* compatible with the ObjectCenter **make** command. Do not use **make -s** in the Workspace.

See Also

build, **clezstart**, **contents**, **load**, **source**

man

displays information about ObjectCenter items

cdm	pdm
✔	✔

Command syntax

man

man *ObjectCenter_item*

Description

<<none>>	Motif and OPEN LOOK: Opens the online *ObjectCenter Reference*. If the *Reference* is already open, scrolls to the first page.
ObjectCenter_item	Motif and OPEN LOOK: Opens the online *ObjectCenter Reference* at the entry for the specified item. If the *Reference* is already open, scrolls to the entry for the specified item.

Usage

Use the **man** command to get online information for ObjectCenter commands and reference topics. All entries in this book, *ObjectCenter Reference*, are included in the online version. Select Manual Browser from the **Browsers** menu or click the "?" button in the Main Window to view the complete online documentation set.

You can also invoke the Manual Browser from a shell with the **cldoc** command.

See Also

english, **help**

memory leak detection

identifying memory leaks

Memory leak detection identifies potential memory leaks by reporting on the memory that the program allocates while running and fails to free before exiting.

The memory leak detection report lists leaks by the size of the memory allocated and identifies the stack trace, indicating where the program allocated the memory. In addition, it shows the number of times the leak occurred there.

Memory leak detection may include pointers to memory that were not freed because the program was exiting. A rule of thumb is that if an allocation is reported more than once, it probably is worth looking at since it may be a real leak.

How to use memory leak detection

To use memory leak detection, follow these instructions:

1. Enter the following in the Workspace in the Main Window of ObjectCenter:

   ```
   setopt mem_trace n
   ```

   The letter *n* represents the maximum number of stack trace levels to report.

2. Run the program in ObjectCenter. Running the program in ObjectCenter creates a file with memory leak detection information when the program exits.

File with memory leak information

For each possible leak, the report file contains two or more lines. The first line has this format:

nbytes [size, count]

where *nbytes* is the total number of bytes, *size* is the memory size allocated each time, and *count* is the number of times the potential leak occurs. Each remaining line contains one level of stack trace in this format:

<tab> *function* **<tab>** *file* **<tab>** *line number*

Example of file from a simple test

Here is a file from a simple test. The file includes a detailed explanation of the contents of the report.

```
# This file contains a listing of possible memory
# leaks : Wed Jan 19 19:00:00 1993
#
# There are 7 possible memory leaks, totaling 19 bytes.
# The format of this report is as follows:
# For each possible leak there are two or more lines. The first
# has the format:
# nBytes [size, count]
# where 'nBytes' is the total number of bytes, 'size' is the
# size allocated each time, and 'count' is how many times it was done.
# Each remaining line for the leak contains one level of stack trace.
# with the format:
# <tab> Function <tab> file <tab> line
#
# (for as many levels of stack trace as requested). #

2       [1, 2]
     main /s/users/smith/Temp/mem2.c 13
2       [2, 1]
     main /s/users/smith/Temp/mem2.c 13
6       [3, 2]
     main /s/users/smith/Temp/mem2.c 13
4       [4, 1]
     main /s/users/smith/Temp/mem2.c 13
5       [5, 1]
     main /s/users/smith/Temp/mem2.c 13
```

Naming your memory detection file

By default, the memory detection file is called **mem.leak** and appears in ObjectCenter's current directory. You can use a different filename by setting the environment variable CENTERLINE_LEAK_FILE to the name you want to use. You must do this before invoking ObjectCenter.

Using ObjectCenter's version of functions

To use ObjectCenter's memory leak detection, you must use ObjectCenter's version of these functions: **malloc()**, **calloc()**, **realloc()**, and **free()**. You cannot substitute your own versions of them.

next

executes source code by line; does not enter functions

cdm	pdm
✔	✔

Command syntax

next

next *number*

Description

<< *none* >>	Executes an entire line, regardless of the number of statements on the line, and then stops execution. Motif and OPEN LOOK: Displays a solid arrow pointing to the current execution line in the Source area.
number	Executes the specified number of lines, and then stops execution.

Options

The following ObjectCenter options affect the **next** command:

cxx_suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.
src_step	(Ascii ObjectCenter only) Specifies number of lines of source code to be displayed after execution of a statement.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **next** command to execute your code line by line without going into functions that are called.

Automatic mode switching

When using **next** to step through code, ObjectCenter automatically matches the Workspace mode to the language type of the module you are currently in. For example, if the current mode is C++ and you use **next** to move into a C module, then ObjectCenter automatically changes to C mode.

Specifying file extensions for C++ source code modules

When you use **next** to move into object code, ObjectCenter needs to find the corresponding source file so that it can display the source code in the Source area. See the **language selection** entry on page 175 for information about how to set the extensions correctly for C++ modules.

The **next** command does not stop inside object code functions that do not have debugging information (functions either compiled without the **-g** switch or loaded with the **-G** switch).

In threaded applications, **next** executes one statement of the specified thread without entering functions.

NOTE Debugging of threaded applications is currently only supported in process debugging mode, and it is not supported on all platforms. Please refer to the "Product limitations" section in the "About This Release" appendix to the online *ObjectCenter Reference*.

Example

This Ascii ObjectCenter example uses a threaded version of the bounce demo from the ObjectCenter tutorial to show how the next command behaves in a threaded application. The drawMove function is executed without stepping into the code. When you issue the next command again, the **doDraw()** function is executed, and **t@6** becomes the new current thread.

```
pdm (break 1) 5 -> thread -info
>   t@5   a l@1 thread_bounce__FP13DrawableShape()running in
doDraw__13DrawableShapeFv()
pdm (break 1) 6 -> next
    112   for (int  count = 0; count < LENGTH; count++) {
pdm (break 1) 8 -> next
    113       drawMove(count);
pdm (break 1) 9 -> thread -info
>   t@5   a l@1 thread_bounce__FP13DrawableShape()running in
doDraw__13DrawableShapeFv()
pdm (break 1) 10 -> next
Breakpoint 1, (t@6,l@4) DrawableShape::bounce(void) (this=0x27c48) at
shapes.C:132
132   doDraw();
pdm (break 1) 11 -> thread -info
>   t@6   a l@4 thread_bounce__FP13DrawableShape()running in
bounce__13DrawableShapeFv()
```

See Also

nexti, **step**, **stepout**

nexti

executes machine code by line; does not enter functions

cdm	pdm
	✔

Command syntax

nexti

nexti *num*

Description

<<*none*>>	Executes the next line of machine code, but does not enter functions.
num	Executes *num* machine instructions, not just the last one, but does not enter functions.

Usage

Use the **nexti** command to step through machine instructions in your program without entering functions.

In threaded applications, **nexti** executes one statement of the specified thread without entering functions.

NOTE Debugging of threaded applications is currently only supported in process debugging mode, and it is not supported on all platforms. Please refer to the "Product limitations" section in the "About This Release" appendix to the online *ObjectCenter Reference*.

See Also

listi, **next**, **stepi**, **stopi**

objectcenter

shell command to invoke the ObjectCenter programming environment

Command syntax

objectcenter [*switches* **]**

objectcenter [*switches* **]** *project_file*

Description

[*switches*] — Use the **-ascii**, **-motif** or **-openlook** switches for the Ascii, Motif or OPEN LOOK versions, respectively.

By default, if you do not specify an interface with any of these three switches, ObjectCenter starts as either Motif or OPEN LOOK, depending on your platform. For example, on Hewlett-Packard workstations, the default GUI is Motif, and for Sun workstations it is OPEN LOOK.

By default, ObjectCenter starts in *component debugging mode*; to start in *process debugging mode*, use the **-**ObjectCenter (process debugging mode) switch.

In component debugging mode, you load all the parts, or components, of your program, and link and execute them within ObjectCenter. In contrast, in process debugging mode, you load your program as a fully linked executable, and you have the choice of debugging it along with a corefile, or attaching to another process. See the **pdm** entry on page 254.

See Table 19 on page 224 for a complete listing of command-line switches you can use with ObjectCenter.

[*switches*] *project_file* — Start ObjectCenter in component debugging mode and load *project_file*.

Usage

The ObjectCenter startup commands are installed in a **CenterLine/bin** directory, which could be installed anywhere on your system. You can start ObjectCenter either by typing the absolute pathname of the ObjectCenter startup command or by putting **CenterLine/bin** on your path and just typing the command name:

```
$ objectcenter
```

See your system administrator if you don't know where **CenterLine/bin** is on your system.

Startup files

When you start ObjectCenter, by default it reads commands from the system-wide startup file, named **ocenterinit**, and from the local startup file, named **.ocenterinit**, which is in your home or current directory.

If you start in process debugging mode, ObjectCenter reads the **.pdminit** file only.

You can use the **-s**[*startup_file*] and the **-S**[*startup_file*] switches to tell ObjectCenter to read *startup_file* instead.

Local startup file

The **.ocenterinit** file is a text file that can contain any input that is accepted in ObjectCenter's Workspace. Typically, you use **.ocenterinit** to set the values of ObjectCenter options and define aliases that are used across ObjectCenter sessions. Since this startup file is read directly into ObjectCenter's Workspace, it should contain only ObjectCenter commands and code that does not need to be debugged or reloaded.

Because ObjectCenter first looks in the current working directory for .ocenterinit, you can have different .ocenterinit files for use with different projects, as long as you work in different directories.

System-wide startup file

The system-wide **ocenterinit** file is in the **CenterLine/configs** directory. Typically, system-wide attributes (such as the directories that ObjectCenter searches for libraries, header files, and so on) are set in **ocenterinit**. If you use different directories than the standard defaults, you need to change the specifications.

You can also use the **ocenterinit** file to set options and aliases for every user at your site.

NOTE ObjectCenter reads the system-wide **ocenterinit** file before the local **.ocenterinit** file, so any specifications in the local file override corresponding specifications in the system-wide file.

Using 8-bit character sets

In component debugging mode (not process debugging mode) ObjectCenter supports 8-bit character sets. Add the following two lines to your local **.ocenterinit** file so you can use the Meta key to get the extended character set:

```
setopt eight_bit
unsetopt line_meta
```

To turn on this feature for all users at your site, ask your system administrator to add these two lines to the global **ocenterinit** file.

Libraries loaded when starting

When starting (in component debugging mode but not process debugging mode), ObjectCenter automatically loads the standard C++ library, **libC.a,** and the standard C library, **libc.a**. On some workstations, ObjectCenter might load the shared versions of these libraries. See the *ObjectCenter Platform Guide* for your platform for any further information about shared libraries.

Switches

This section describes the ObjectCenter command-line switches.

ObjectCenter processes command-line switches in the order in which they are specified.

You can use most of the switches with any version of ObjectCenter, although a couple are specific to Ascii ObjectCenter. See Table 19 for an alphabetical listing and description of all the switches.

NOTE In addition to command-line switches, ObjectCenter supports many options that you can use to control its features and commands. See the **options** reference page for more information.

Table 19 Command-Line Switches Supported by ObjectCenter

Name of Switch	What the Switch Does	Restrictions
-ascii	Start ObjectCenter with the character-based rather than a graphical user interface. Do not use this switch with **-openlook** or **-motif**.	None.
-backend_ansi	Sets the **backend_ansi** option, which makes ObjectCenter generate ANSI C intermediate code. By default, K&R C intermediate code is generated. See the **code generation** entry on page 91.	Has no effect in pdm.
-class_as_struct	Disables maximum processing of classes to improve performance.	No effect in .
-d	(**load** switch). Turns off terminal-dependent output; as a result, raw mode input will be disabled so that pressing the Return key is required to respond to a prompt.	Has no effect in pdm.
-D*name[=def]*	(**load** switch). Causes name to become defined as if a **#define** directive had occurred. If *def* is not supplied, then 1 is used. For more information, see the **load** entry .	Has no effect in pdm.
-f *log_name*	Saves a copy of all input typed in the Workspace in a permanent file called *log_name*. All input is usually saved in a temporary logfile that is deleted when you quit ObjectCenter. Note that a space is required between the switch and the argument for the switch.	None.
-full_symbols	Forces the reading of the full symbol table for maximum information immediately.	Has no effect in cdm.
-G	(**load** switch). Ignores debugging information produced by the **-g** switch of the compiler when loading compiled files. For more information, see the **load** entry .	Has no effect in pdm.
-I*header_path*	(**load** switch). Adds *directory_name* to the list of directories to search for files specified by the **#include** preprocessor directive. For more information, see the **load** entry .	Has no effect in pdm.
-i *input_file*	Specifies that ObjectCenter's command input should be read from *input_file*, rather than from standard input. Note that a space is required between the switch and the argument for the switch.	Has no effect in pdm.

Table 19 Command-Line Switches Supported by ObjectCenter (Continued)

Name of Switch	What the Switch Does	Restrictions
-L*library_path*	(**load** switch). Adds *library_path* to the list of directories to search for libraries. For more information, see the **load** entry on page 185.	Has no effect in pdm.
-l*lib_name*	(**load** switch). When a library is loaded using the **-l** *x* format, then a file called **lib***x***.a** is sought first in the directories specified by **-L** options, then in the standard directories /**lib**, /**usr**/**lib**, and /**usr**/**local**/**lib**. For more information, see the **load** entry on page 185.	Has no effect in pdm.
-m *target*	Indicates that ObjectCenter should perform a **make** on *target* when starting up. This is equivalent to entering **make** target as the first statement in the Workspace. Note that a space is required between the switch and the argument for the switch.	Has no effect in pdm.
-motif	Start ObjectCenter with the Motif graphical user interface. Do not use this switch with **-openlook** or **-ascii**.	None.
-no_fork	Create a separate Run Window but avoid returning immediate control to the shell. With **-no_fork**, control returns when you enter ^Z in the shell or exit ObjectCenter. Without **-no_fork**, the shell prompt comes back immediately.	None.
-no_run_window	Avoids creating the separate Run Window and avoids returning control to the shell. Your program's output goes to the shell in which you invoked ObjectCenter. Using the **-no_run_window** switch means you are unable to interrupt ObjectCenter and unable to place it in the background. This switch is intended for debugging applications that need specific terminal support rather than a generic terminal such as xterm.	None
-o *output_file*	Specifies that ObjectCenter's command output should be written to *output_file*, rather than to standard output. Note that a space is required between the switch and the argument for the switch.	Has no effect in pdm.
-openlook	Start ObjectCenter with the OPEN LOOK graphical user interface. Do not use this switch with **-motif** or **-ascii**.	None.

Table 19 Command-Line Switches Supported by ObjectCenter (Continued)

Name of Switch	What the Switch Does	Restrictions
-pdm	Start ObjectCenter in process debugging mode. See the **pdm** entry on page 254 for more information.	None.
-r *number*	Specifies the size of the run-time stack as *number* nested function calls. The default size is approximately 1000 nested function calls. The default size may need to be increased when executing highly recursive programs. Note that a space is required between the switch and the argument for the switch.	Has no effect in pdm.
-S *[startup_file]*	If *startup_file* is supplied, it is read at startup instead of the default system startup file. If a file name is not specified, the system startup file is ignored. Use a dash (**-**) to indicate that there is no startup file. Note that a space is required between the switch and the argument for the switch.	None.
-s *[startup_file]*	If startup_file is supplied, it is read at startup instead of the **.ocenterinit** file, which is the default startup file. If a file name is not specified, the default startup file is ignored. Use a dash (**-**) to indicate that there is no startup file. Note that a space is required between the switch and the argument for the switch.	None.
-softbench	Starts up **clms_gateway** and places ObjectCenter in the SoftBench Tool Manager's window.	HP only.
-U*macro_name*	(**load** switch). Causes the predefined *macro_name* to become undefined as if a **#undef** directive had occurred.	Has no effect in pdm.
-usage	Displays a table of switch abbreviations and arguments.	None.
-w	(**load** switch). Suppresses reporting of warnings; errors are always reported. For more information, see the **load** entry on page 185.	Has no effect in pdm.

NOTE ObjectCenter no longer supports the **-p** switch. By default, all file descriptors that are open when you start ObjectCenter remain open during your session.

See Table 20 for a list of switches that you can use with the **objectcenter** command along with the **-motif** or **-openlook** switches. These command-line switches allow you to fine tune your windowing environment without having to edit any files specifying X resources.

Table 20 Switches to Specify Graphical User Interface from Command Line

Name of Switch and Arguments, if any	What the Switch Does
-background *color* **-bg** *color*	Specifies background color.
-config *pathname*	Uses the X resource specifications in *pathname* instead of the defaults. See the **X resources** entry on page 436 for more information.
-debug	Enables protocol error handler. If this switch is specified, any X protocol error or fatal OI™ error causes an error message to be printed on **stderr** followed by a core dump. Note that running ObjectCenter with a command line of **-debug** is different than compiling with a flag of **-debug**. If the command-line argument **-debug** is not specified, the error messages still print when these errors occur, but a core dump is not produced.
-display *host:dpy.scn*	Specifies X server to connect to. If **-display** *host:dpy.scn* is specified, the program's display is targeted for machine *host* on the network, on display and screen *dpy.scn*. If this argument is not specified, the display is taken from the environment variable **DISPLAY**, if it exists; otherwise, the display is targeted for the originating host, display and screen using **unix**:**0.0**.
-fastdraw	Tells ObjectCenter to sacrifice appearance for faster drawing. If **-fastdraw** is specified, the appearance of objects drawn on the screen will be compromised for faster drawing. This is useful if your program is displaying on an X terminal over an RS232 line.
-font *font_name*	Specifies default text font for all objects in the GUI.
-foreground *color* **-fg***color*	Specifies foreground color.
-iconic	Tells ObjectCenter to start in iconic state.

Table 20 Switches to Specify Graphical User Interface from Command Line (Continued)

Name of Switch and Arguments, if any	What the Switch Does
-name *name_string*	Specifies *name_string* as the name for this instance of program. If **-name** *name_string* is specified, *name_string* will be the value of the instance portion of the **WM_CLASS** property for this instance of the execution of the program. If **-title** is not also specified, *name_string* will be the value of the **WM_NAME** property, and will be displayed in the title bar of the main application window (assuming the window manager uses **WM_NAME**).
-ol	Tells ObjectCenter to use the most appropriate OPEN LOOK model. If the monitor is monochrome, the 2-D model is used; if the monitor is color, the 3-D model is used.
-ol2d	Tells ObjectCenter to use the 2-D OPEN LOOK model.
-openlook_2d	
-ol3d	Tells ObjectCenter to use the 3-D OPEN LOOK model.
-openlook_3d	
-reverse	Tells ObjectCenter to reverse foreground and background colors.
-rv	
-xrm‘*resource_string:value*’	Sets the X resource *resource_string* in the X resource database to the *value* string.

NOTE See the X resources reference page for more information about setting up your X environment.

options

Many of ObjectCenter's commands and windowing features are controlled by options. Most of these options are only available in component debugging mode. The tables in this entry indicate which options are available in process debugging mode.

Use the following ObjectCenter commands to manipulate options:

setopt Sets the values of options.

printopt Displays the values of options.

unsetopt Unsets the values of options

NOTE You can also use ObjectCenter's Options Browser to display and change options. In general, using the Options Browser is the best and easiest way to set options. See the *User's Guide* for a description of the Options Browser**.**

Functional summary of the options

See Table 21 for a summarized list of the options arranged according to the following functional categories:

- Editor control
- Environment control
- Information lookup
- Language control
- Listing control
- Load control
- Make control
- Memory control
- Output control
- Paging control
- Run control
- Window control
- pdm options

Since some options apply to more than one category, they are listed more than once.

NOTE Table 22, later in this entry, lists all ObjectCenter options alphabetically, along with the ObjectCenter commands that are affected by each option, the data type of the option, the default value of the option, and a description of what the option does.

Table 21 ObjectCenter Options Summarized According to Functional Category

Functional Category	Name of Option	Brief Description
Editor control (Ascii CodeCenter only)	**editor**	Set this option only if there is no edit server in the environment.
Environment control	**centerline_path**	Specifies search path for executables, documentation files, and other files.
	email_address	Specifies the electronic mail address for the **email** command.
	eight_bit	Tells ObjectCenter to treat input and output as 8-bit characters.
	line_edit	Adds line editing, command completion, and extensive history capabilities to the Workspace.
	line_meta	Lets all 8 bits pass as input.
	logfile	Specifies the name of the file used to record all Workspace input.
	path	Specifies the search path for loading source and object files (not for **#include** files).
	proto_path	Specifies the search path for prototype files supplied by CenterLine.
	shell	Specifies the shell that is started by the **shell** or the **#!** command.

Table 21 ObjectCenter Options Summarized According to Functional Category (Continued)

Functional Category	Name of Option	Brief Description
	subshell	Specifies the shell used to invoke the C++ translator or C compiler.
Information lookup	**support_phone**	Specifies local customer support phone number.
	version_date	Specifies the date that the ObjectCenter software was released.
	version_number	Specifies the version number of ObjectCenter.
	workgroup_id	Specifies the workgroup ID for your license of ObjectCenter.
Language control	**ansi**	Performs preprocessing in strict conformance with the ANSI C Standard.
	backend_ansi	Makes ObjectCenter generate ANSI C intermediate code. By default, ObjectCenter generates K&R C intermediate code. See the **code generation** entry on page 91.
	primary_language	Specifies the language ObjectCenter uses for interpreting a module if you do not specify **-C** or **-CXX**. See the **language selection** entry on page 175 for more information.
Listing control	**src_err**	Specifies the number of source lines to be listed for errors and warnings (Ascii ObjectCenter only).
	src_step	Specifies the number of lines of source code to be displayed after execution of a statement (Ascii ObjectCenter only).
	src_stop	Specifies the number of lines of source code to be displayed when a break level is first created (Ascii ObjectCenter only).
Load control	**auto_compile**	Automatically compiles missing or outdated object files.

Table 21 ObjectCenter Options Summarized According to Functional Category (Continued)

Functional Category	Name of Option	Brief Description
	batch_load	Suppresses prompts to the user during loading.
	cc_prog	Specifies the name of the C compiler that ObjectCenter invokes.
	ccargs	Specifies arguments passed to **cc** when invoked from ObjectCenter. Note that when you use **make**, if the environment variable CFLAGS is set, it takes precedence over **ccargs**.
	create_file	Specifies commands to create a new file when loading.
	cxx_prog	Specifies name of C++ translator to be invoked by ObjectCenter when compiling a C++ file.
	c_suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C source file that corresponds to a given object file. See the **language selection** entry on page 175 for details about configuring the language used by ObjectCenter.
	cxx_suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file. See the **language selection** entry on page 175 for details about configuring the language used by ObjectCenter.
	cxxargs	Specifies default arguments passed to the C++ translator when invoked by ObjectCenter.
	echo	Echoes the input stream after preprocessing (similar to the **-E** compiler switch).

Table 21 ObjectCenter Options Summarized According to Functional Category (Continued)

Functional Category	Name of Option	Brief Description
	instrument_all	Automatically instruments files as they are loaded. See the **instrument** entry on page 162 for more information about instrumenting.
	instrument_byte	Checks for uninitialized memory that is used one byte at a time. See the **instrument** entry on page 162 for an example.
	instrument_space	Allocates memory required to instrument files. See the **instrument** entry on page 162.
	load_flags	Specifies the default switches to use if **load** is called without any switches.
	page_load	Sets the number of lines of error reports to display before prompting the user for more.
	path	Specifies the search path for loading source and object files (not for **#include** files.)
	preprocessor	Specifies a command to execute in a subshell before the file is loaded.
	proto_path	Specifies the search path for prototype files that you supply.
	src_err	Specifies the number of source lines to be listed for errors and warnings.
	subshell	Specifies the shell used to invoke the C++ translator or C compiler.
	sys_load_cflags	Specifies switches that establish the search path for system libraries and include files when loading C files(files loaded with **-C**).
	sys_load_cxxflags	Specifies the switches that establish the search path for system libraries and include files when loading a C++ file.
	tmpl_instantiate_flg	Specifies switches related to template instantiation; see Table 30 on page 360.

Table 21 ObjectCenter Options Summarized According to Functional Category (Continued)

Functional Category	Name of Option	Brief Description
	tmpl_instantiate_obj	Tells ObjectCenter to load instantiation modules for templates as object rather than source.
Make control	**auto_compile**	Automatically compiles missing or outdated object files.
	make_args	Specifies the command-line arguments passed to the UNIX **make** utility by ObjectCenter's **make** command.
	make_hfiles	Checks header files to find out if a file should be reloaded.
	make_offset	Specifies the number of characters to skip when reading shell commands from the **make** program.
	make_prog	Specifies the program invoked to make a target. (The default is **make**.)
	make_symbol	Specifies the string used to denote an ObjectCenter command line in a makefile. Use only if you also specify **old_make**.
	subshell	Specifies the shell used to invoke the C++ translator or C compiler.
Memory control	**mem_config**	Tunes the memory allocator to optimize memory usage.
	mem_trace	Specifies the level of memory tracing to perform.
	save_memory	Set this option if memory is scarce or for portions of a program that allocate very large arrays.
	sbrk_size	Specifies the amount of memory that can be allocated by the **sbrk()** and**brk()** system calls.

Table 21 ObjectCenter Options Summarized According to Functional Category (Continued)

Functional Category	Name of Option	Brief Description
	unset_value	If set to **0**, tells ObjectCenter not to report variables used without being set.
Output control	**list_action**	Displays actions that execute everywhere when listing the source line at which they were triggered.
	print_inherited	Filters the display of inherited data members.
	print_pointer	Adds diagnostic information to pointer display.
	print_runtime_type	Specifies pointer display as run-time rather than compile-time types.
	print_static	Specifies the display of static data members.
	print_string	Specifies the number of characters of a string to print.
	show_inheritance	Tells ObjectCenter to show full rather than truncated inheritance path.
	tab_stop	Specifies the number of spaces to indent per tab character when listing source code.
	terse_suppress	Tells the **suppress** command not to echo the name of the violation being suppressed.
	terse_where	Tells the **where** command not to list the formal arguments of each function on the execution stack.
Paging control	**page_cmds**	Sets the number of lines of output displayed before a **more** prompt is issued.
	page_list	Sets the number of lines of source code the **list** command displays before a **more** prompt is issued.
	page_load	Sets the number of lines of error reports to display before prompting the user for more.

Table 21 ObjectCenter Options Summarized According to Functional Category (Continued)

Functional Category	Name of Option	Brief Description
Run control	**batch_run**	Specifies the method for handling run-time violations.
	lint_run	Indicates the severity of warnings issued by ObjectCenter during execution.
	program_name	Specifies the value of the first argument, **argv[0]**, to **main()**.
Window control	**win_fork**	If set, a new window is created when a program forks. In Ascii ObjectCenter, prompts for a new tty device. All input and output to this new child process will take place in the new window.
	win_io	Directs output to the Workspace if unset. (We recommend that you keep this option set for complicated programs that use curses-style input and output.)
	win_no_raise	Prevents deiconifying the Run Window when you issue the run or start command. The default behavior is to deiconify the Run Window.
pdm options	**class_as_struct**	Disables maximum processing of classes to improve performance
	full_symbols	Forces the reading of the full symbol table for maximum information immediately.

NOTE The following options, which were available in previous releases of ObjectCenter, are no longer available: **auto_reload**, **auto_replace, centerline_port**, **debug_child, num_proc**, **term**, **win_fork_nodup, win_project_list, win_message_list**.

Table 22 lists all ObjectCenter options alphabetically, along with the ObjectCenter commands that are affected by each option, the data type of the option, the default value of the option, and a description of what the option does.

Table 22 ObjectCenter Options

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
ansi	Boolean	unset (FALSE)	**load**
	Perform preprocessing in strict conformance with the ANSI C Standard. This option also applies the ANSI C Standard to any C code entered in the Workspace under **cmode**. See the **ANSI C** entry on page 12 for more information about ANSI C and ObjectCenter; also see the **config_c_parser** entry on page 103.		
auto_compile (Ascii CodeCenter only)	Boolean	set (TRUE)	**build** **load**
	Automatically compile missing or outdated object files when **load** is invoked. If you invoke a **build** from the Project Browser, this option is ignored; missing or out-of-date files are always recompiled. If a makefile exists, **load** calls **make** *args file*, where *args* is specified by the **make_args** option. If a makefile does not exist, **load** uses the command for C++ files, **CC** *args file*, where *args* is specified by the **cxxargs** option; for C files, **cc** *args file*, where *args* is specified by the **ccargs** option.		
backend_ansi	Boolean	unset (FALSE)	**build** **link** **load**
	Makes ObjectCenter generate ANSI C intermediate code. By default, ObjectCenter generates K&R C intermediate code. For more information, see the **code generation** entry on page 91.		
batch_load (Ascii CodeCenter only)	Boolean	unset (FALSE)	**load**
	Do not prompt with options when warnings are encountered during the loading process. Prompt to continue only if more than **page_load** lines of messages are displayed. See the **page_load** option.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
batch_run	Integer	**0**	**rerun** **start** **run**
	Proceed as follows when a run-time violation is detected, according to the appropriate value. 0 Stop and issue a prompt at each run-time warning or error. 1 Record all warnings and continue, but prompt at each error. 2 Record all warnings and continue, record the first error, then stop execution. 3 Record all warnings and errors and continue. Note that this setting can be dangerous, since errors can cascade. Do not use this option when you have unresolved references.		
cc_prog	String	**cc**	**load** **make**
	Use the C compiler with the name specified.		
ccargs	String	**-g**	**build** **load** **make**
	Pass the specified arguments to **cc** when **cc** is invoked from ObjectCenter. ObjectCenter invokes **cc** directly, using **ccargs**, only if it needs to recompile a file that does not have a makefile; if a makefile is available, ObjectCenter invokes **make**.		
centerline_path	String	determined when you start ObjectCenter	none
	Use the specified search path for executables, documentation files, and various other files.		
class_as_struct	Boolean	unset (FALSE)	**debug**
	Disables maximum processing of classes to improve performance. (This option is only available in pdm.)		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
create_file	String with delimiter	null	**load**

Execute the command specified by *string* to create a file if **load** specifies a file that does not exist. *string* has the following format:

@file1@command1@file2@command2...@*@commandx

where:

@ is a delimiter. The delimiter can be any character; however, avoid using the characters * and % since they are meta-characters for this command. Use the delimiter at the beginning of the string and between items in the string.

file1 is an argument to the **load** command. If *file1* does not exist, *command1* is used to create *file1*.

file2 is similar to *file1*. If *file2* does not exist and is specified, then *command2* is used to create *file2*, and so on.

* If the specified file does not match any *fileN* in *string*, *commandx* is used; the * matches all files. When you specify *commandx*, use **%s** to reference the filename.

For example, the following *string* is a possible value for **create_file**:

```
@foo.c@yacc foo.y
@bar.c@yacc bar.y
@*@co -l %s
```

Given this value for **create_file**, suppose you issue the following command:

```
load foo.c
```

If **foo.c** does not exist, ObjectCenter invokes **yacc foo.y**.

Similarly, if the file specified is **bar.c**, ObjectCenter invokes **yacc bar.y**.

Suppose the file specified with **load** is neither **foo.c** nor **bar.c**:

```
load notfoobar.c
```

In this case, ObjectCenter invokes the following command:

```
co -l notfoobar.c
```

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
c_suffixes			**build** **edit** **list** **load** **next** **step** **swap**

Specifies file extensions to search for when ObjectCenter needs to find a C source file that corresponds to a given object file. This need arises under the following conditions:

- You issue the **load** command and specify an object file. In this case, ObjectCenter needs to check the corresponding source file to see if the object file is up-to-date. If not, ObjectCenter compiles a new object file before proceeding.
- You issue the **build** command with an object module loaded. In this case, ObjectCenter needs to check the corresponding source file to see if the object file is up-to-date. If not, ObjectCenter compiles a new object file before proceeding.
- You issue the **swap** command and specify an object file that is loaded. In this case, ObjectCenter searches for the corresponding source file, unloads the object file, then loads the source file.
- You step through object code. In this case, ObjectCenter needs to locate the corresponding source file to display the source code.
- You issue the **list** or the **edit** command and specify a function loaded in object form. In either of these cases, ObjectCenter needs to locate the corresponding source file to display the source code.

See the **language selection** entry on page 175 for details about configuring the language used by ObjectCenter.

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
cxxargs	String		**build** **load** **make**
	Specifies the default arguments passed to the C++ translator when invoked by ObjectCenter. The default arguments are **+d -g**. The **+d** switch specifies that inline functions should not be expanded. The **-g** specifies that debugging information should be included in the object file.		
cxx_prog	String	unset, which means that ObjectCenter invokes **CC**.	**load** **make**
	Specifies the name of the C++ translator that ObjectCenter invokes when compiling a C++ file. The default is unset, which means that ObjectCenter invokes **CC**.		
cxx_suffixes			**build** **edit** **list** **load** **next** **step** **swap**
	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file. For more information about when this need arises, see the **c_suffixes** option described on page 240. See the **language selection** entry on page 175 for details about configuring the language used by ObjectCenter.		
echo	Boolean	unset (FALSE)	**load**
	Echo the input stream after preprocessing has taken place. This result is similar to calling the compiler with the **-E** switch.		
editor (Ascii CodeCenter only)	String	**vi**	**edit**
	By default this option is unset. Set it only if there is no edit server in your environment. Possible values are **vi** and **emacs**.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
eight_bit	Boolean	unset (FALSE)	**keybind**
	Treat input and output as 8-bit characters.		
email_address	String	**objectcenter_ support@ centerline.com**	**email**
	Specify the electronic mail address for the **email** command. The option is set in the **CenterLine/configs/support-defs** file.		
full_symbols	Boolean	unset (FALSE)	**debug**
	Forces the reading of the full symbol table for maximum information immediately. (This option is only available in pdm.)		
ignore_sharp_lines	Boolean	unset (FALSE)	**load**
	Ignore **#line** directives generated by preprocessors. Consequently, ObjectCenter does not maintain any correspondence between a preprocessor input file and a source code output file that is currently loaded. This option does not affect object files.		
instrument_all	Boolean	unset (FALSE)	**load**
	Automatically instrument files as they are loaded. See the **instrument** entry on page 162 for more information.		
instrument_byte	Boolean	unset (FALSE)	**instrument**
	Check for unset memory that is used one byte at a time. See the **instrument** entry on page 162 for an example.		
instrument_space	Integer	2	**instrument**
	Allocate space as specified for instrumented object code. The default value of **2** allows an amount of space approximately 50% of the text size of your application. If you get a message that more space is needed, we recommend you increase the value of this option by 1 until you have allocated enough space. If you set the value of this option to **0**, you save space, but you cannot instrument any object files.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
line_edit	Boolean	set (TRUE)	**history** **keybind**
	Add line editing, command completion, and extensive history capabilities to the Workspace. Also, with this option set, some errors in the Workspace echo a bad input line again to facilitate easy correction. If unset, key bindings have no effect.		
line_meta	Boolean	unset (FALSE) *except* on Sun workstations, where the default is set (TRUE)	**keybind**
	Let all 8 bits pass as input to ObjectCenter. On Sun keyboards, this allows the meta key (marked ◊ or **Left** or **Right**) to be treated as a Meta key. **0** suppress all warnings **1** report all violations, including **lint**-style warnings		
lint_run	Integer	2	**rerun** **run** **start**
	Issue run-time warnings at the severity specified. The possible settings are as follows: **0** suppress all warnings **1** suppress minor warnings (such as type mismatch during function calls) **2** report all possible violations		
list_action (Ascii CodeCenter only)	Boolean	set (TRUE)	**action** **list**
	Display actions that execute everywhere when listing the source line at which they were triggered.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
load_flags	String	null	**load**
	Use the specified switches as the default switches but only if **load** is called without any switches. ObjectCenter always uses any **-L** switches specified in **load_flags**. See the "When does the load_flags option have precedence?" **TIP** on page 191.		
logfile	String	a temporary filename	none
	Use the specified file to record all your Workspace input. The file is periodically used by ObjectCenter, so be sure not to delete it.		
make_args	String		**build** **make**
	Pass the specified command-line arguments from ObjectCenter's **make** command to the UNIX **make** utility. These could be arguments such as **-DOBJECTCENTER_MAKE** or **-f** *filename*.		
make_hfiles	Boolean	set (TRUE)	**build**
	Check header files to determine whether a file should be reloaded. If you are loading a large project, setting this option can be time consuming. This option has no effect in pdm.		
make_offset	Integer		**make**
	Skip the number of characters specified when reading shell commands from the **make** program. This option has no effect in pdm.		
make_prog	String	**make** (UNIX command)	**make**
	Invoke the specified program to make a target.		
make_symbol	String	**#**	**make**
	Use the specified string to denote an ObjectCenter command line in a makefile. This option has no effect in pdm.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
mem_config	Integer	16384	none
	Tell ObjectCenter how much memory to allocate at a time from the operating system. The default sets memory to twice the average large **malloc()** size. Use this option to tune the memory allocator to optimize memory usage for a particular application.		
mem_trace	Integer	0	none
	Specifies the level of memory tracing to perform. (0 is off). Tell ObjectCenter to write potential memory leak information to a file called **mem.leak** in the current working directory when the application being run exits. All memory that is allocated while a program is running and not freed before the program exits is reported. For more information see the **memory leak detection** entry on page 216.		
page_cmds (Ascii CodeCenter only)	Integer	the size of the terminal's screen	none
	Set the number of lines of output from commands that are displayed before issuing a **more** prompt. If unset, command output is not paginated.		
page_list	Integer	10	**list**
	Set **list** command to display the specified number of lines of source code before issuing a **more** prompt. If this option is unset, ObjectCenter does not paginate source code listings.		
page_load (Ascii CodeCenter only)	Integer	the size of the terminal's screen	**load**
	Specify the number of lines of error reports to display before prompting the user for more. If this option is unset, ObjectCenter displays a screenful. This option is meaningful only if used along with **batch_load**.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
path	String	null	**cd** **edit** **list** **load** **swap**
	Search the directories in the order specified when the affected commands are invoked. The current directory is always added implicitly to the end of the path. If this option is unset, which is the default, ObjectCenter searches the current directory. You must also set the **swap_uses_path** option for **path** to affect the **swap** command. Separate the directory names by spaces; you can specify the directories as absolute or relative pathnames. The *path* option does not provide a search path for loading **#include** files or libraries — only for loading source and object files and for a matching pathname with the **cd** command. To give the search path for **#include** directories, use the **-I** switch with the **load** command according to the following format: -> **load -I***include_dir1* **[-I***include_dir2* **...]** *file.c* See the “Loading libraries” section on page 195 and also the "Specifying the search path for loading libraries and #include files" **TIP** on page 196. You can set the value of the **path** option with either the **use** command or the **setopt** command.		
preprocessor	String	null	**load**
	Execute the specified command in a subshell before loading the file. The command should have a **%s** in it, which is replaced by the name of the file being loaded. The output from the subshell is loaded. If the command changes the number of lines in the file, then subsequent references to source lines will not be accurate. The following example specifies **m4** as a preprocessor: `-> setopt preprocessor m4 macro_file %s`		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
primary_language	String	C++	
	Designates either C or C++ as the default language for the programing environment.		
print_inherited	Boolean	set (TRUE)	**display** **print**
	Filters the display of inherited data members. If set / TRUE, inherited data members are displayed. If unset / FALSE, inherited data members are hidden.		
print_pointer	Boolean	set (TRUE)	**display** **print**
	Display pointers with diagnostic information about what they point to.		
print_runtime_type	Boolean	set (TRUE)	**display** **print**
	If set, displays the run-time type of the object being pointed to when dereferencing a pointer or displaying a reference. If unset, displays the definition-time (“compile-time”) type of the object being pointed to		
print_static	Boolean	unset (FALSE)	**display** **print**
	Specifies the display of static data members. If set / TRUE, static data members are displayed. If unset / FALSE, static data members are hidden.		
print_string	Integer	20	none
	Use the number specified as the number of characters of a string to print. Use an ellipsis (**...**) following the string if more characters can be displayed.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
program_name	String	**a.out**	**rerun** **run** **start**
	Use the value specified as the value of the first argument, **argv[0]**, to **main()**. This option is especially useful for X11™ applications, which often rely on the program name for resource setting. In X11, resources are looked up relative to the program name, which by default is **a.out** in ObjectCenter. If you want to change the default program name to the one you usually use to invoke your application, set the **program_name** option with the **setopt** command.		
proto_path	String	none	**load**
	Look in the specified directory for prototype files. When loading, if a file ends in .**proto**, the directories specified by **proto_path** are searched for the named file. The format for this option is a list of directories separated by spaces. For best results, use absolute pathnames. **NOTE:** ObjectCenter no longer provides prototype files; however, you can use this option to specify the directory containing ones that you provide yourself.		
save_memory	Boolean	unset (FALSE)	**action**
	Have library functions **malloc()** and **calloc()** not use **centerline_malloct()** to allocate memory. Consequently, ObjectCenter does not use extra memory for run-time type checking when the program allocates memory. Set this option if memory is scarce or for portions of a program that allocate very large arrays. **NOTE:** When **save_memory** is set, run-time warnings such as dynamic type mismatch and dynamic used-before-set are not reported. Watchpoints and actions cannot be set on dynamic memory if **save_memory** is set.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
sbrk_size	Integer	1048576 bytes	none
	Use the specified amount of memory as the amount that can be allocated by the **sbrk()** and **brk()** system calls. This option must be set before a program's first use of **sbrk()**; do not make the first allocation with **brk()** in ObjectCenter. Programs that use only the **malloc()** functions are not affected by this option. Only programs that call the **sbrk()** system calls directly are restricted to allocating the amount of memory specified by this option; these are usually programs that contain their own memory allocation routines. The upper limit for this option is usually determined by the amount of swap space available on the system.		
shell	String	**CENTERLINE_ SHELL**(if it exists; otherwise **SHELL** environment variable)	**shell**
	Start up the specified shell when invoked by the **shell** or the **#!** command. At startup, **shell** is set to the value of the environment variable **CENTERLINE_SHELL**, if it exists; otherwise, **shell** is set to the environment variable **SHELL**. **NOTE:** This option does not affect the **sh** command.		
show_inheritance	Boolean	set (TRUE)	**browse_ class** **browse_data_ members** **browse_friends** **browse_member_functions** **classinfo, display, print**
	Display class members with the full inheritance path showing how members are inherited. If unset, show a truncated inheritance path giving only the defining class. In the inheritance path listing, single colons (:) indicate inheritance and the scoping operator (::) indicates the class that defines a member.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
src_err (Ascii CodeCenter only)	Integer	3	**load**
	List the specified number of source lines when an error or warning is reported.		
src_step (Ascii CodeCenter only)	Integer	1	**next** **step**
	Specify number of lines of source code the **step** and **next** command display after stepped execution of a statement.		
src_stop (Ascii CodeCenter only)	Integer	3	**stop**
	Display the specified number of code lines when a break level is created for the first time.		
subshell	String	**/bin/sh**	**load** **make**
	Use the shell specified to invoke the C++ translator C compiler.		
support_phone	String	local customer support phone number	none
	Specify local customer support phone number.		
swap_uses_path	Boolean	unset (FALSE)	**swap**
	Use the **path** option when looking for files. If this option is unset, which is the default: The **swap** command does not look in the directories specified in the **path** option to find a file.The **swap** command looks only in the same directory as the file being swapped out. If the file to swap in is in a different directory than the file being swapped out, the swap fails. If you are swapping from source to object, ObjectCenter has the source file compiled in the same directory and loads the new object file.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
	If this option is set, *and* you are swapping from source to object: The **swap** command first looks in the directory of the source file. If there is no corresponding object file in that directory, **swap** then follows the search path set by the **path** option and loads the first corresponding object file it encounters. If a corresponding object file is not in any of the directories searched, **swap** has the source file compiled and loads the resulting object file. If this option is set, *and* you are swapping from object to source: The **swap** command first looks in the directory of the object file being swapped out. If there is no corresponding source file in that directory, **swap** then follows the search path set by the **path** option and loads the first corresponding source file it encounters.		
sys_load_cflags	String		**load**
	Use the specified switches to establish the search path for system libraries and **#include** files. These switches are always passed to the *load* command when loading C source files (files loaded with **-C**).		
sys_load_cxxflags	String		**load**
	Specifies the switches that establish the search path for system libraries and **#include** files. These switches are always passed to the *load* command when loading a C++ file.		
tab_stop	Integer	8	**list**
	Indent the number of spaces specified per tab character when listing source code.		
terse_suppress	Boolean	unset (FALSE)	**suppress**
	Set the **suppress** command not to echo the name of the violation being suppressed.		
terse_where	Boolean	unset (FALSE))	**where**
	Set the **where** command not to list the formal arguments of each function on the execution stack.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
tmpl_instantiate_flg	String		
	Specifies the switches to be used when instantiating templates. See the "Using options to control the template instantiation process" **TIP** on page 363 for more information.		
tmpl_instantiate_obj	Boolean	set (TRUE)	
	Tells ObjectCenter to load instantiation modules for templates as object rather than source.		
unset_value	Integer	**191**	**instrument**
	Use this value to detect memory that has not been set. Every byte of memory allocated by the **malloc()** functions and by **centerline_unset()**, as well as memory for automatic variables, is set to the value of **unset_value**. If this option is set to **0** (**setopt unset_value 0**), ObjectCenter no longer diagnoses that a variable is used without being set. If your program happens to set the value of memory to match the value of this option, ObjectCenter probably generates spurious run-time warnings. One way to eliminate them is to change the value of **unset_value**.		
version_date	String	current version date	none
	Display the date the ObjectCenter software was released.		
version_number	String	the current version	none
	Display the version number of ObjectCenter.		
win_fork	Boolean	set (TRUE)	none
	If set, a new window is created when a program forks. In Ascii ObjectCenter, prompts for a new tty device. All input and output to this new child process will take place in the new window.		

Table 22 ObjectCenter Options (Continued)

Name of Option	Type	Default Value	Commands Affected
	What the Option Tells ObjectCenter To Do		
win_io	Boolean	set (TRUE)	none
	Directs output to the Workspace. Your output will go to the Workspace at your next reinit, whether it is an implicit reinit (for example, when you issue the **run** command) or an explicit reinit (by issuing the **reinit** command). We recommend that you keep the **win_io** option set, however, for complicated programs that use curses-style input and output. Unsetting **win_io** has the following limitations: • Controlling-tty semantics are unavailable in the Workspace. This means that **tcgetpgrp/tcsetpgrp** and **tty**-generated signals will not work as expected. • If your program affects the **tty** mode, it may affect the Workspace output. • The tty mode may not be preserved across Workspace interactions. For example, when you continue from a breakpoint, the **tty** settings may not be the same as when you stopped.		
win_no_raise	Boolean	unset (FALSE)	none
	Prevents deiconifying the Run Window when you issue the **run** or **start** command. The default behavior is to deiconify the Run Window.		
workgroup_id	Integer		none
	Display the workgroup ID for your license of ObjectCenter.		
workspace_include	Boolean	unset (FALSE)	none
	Enables the use of **#include** in the Workspace. This option is provided for backwards compatibility with earlier releases. We recommend that you use the **load_header** command to load definitions from header files.		

pdm

process debugging mode; used for debugging an executable file, a corefile, or a running process

Purpose of pdm

Using ObjectCenter's pdm, or process debugging mode, allows you to examine what is going on in a program while it executes. You can use process debugging mode to debug an executable file (**a.out**) along with a corefile or a running process. A corefile contains a literal copy of the contents of memory at the time that the operating system aborted a program.

You can use pdm to do the following:

- Start your program under varying conditions that might affect its behavior
- Stop your program on specified conditions
- See what has happened when your program has stopped
- Change your program, so you can try out solutions to problems you discover

Note that you *cannot* use pdm for automatic load-time or run-time error checking; see the **debugging** entry on page 116 for an overview of these other forms of debugging supported by ObjectCenter.

Invoking pdm

There are several different ways to invoke ObjectCenter's process debugging mode, depending on whether or not you are already in the ObjectCenter environment.

Outside the ObjectCenter environment

If you are not already in the ObjectCenter environment, you can start ObjectCenter in process debugging mode by using the **-pdm** switch on the shell command line:

```
$ objectcenter -pdm
```

When you start pdm, ObjectCenter adds **CenterLine/***arch-os***/lib** to your LD_LIBRARY_PATH environment variable so that it can find required shared libraries.

NOTE The **objectcenter -pdm** shell command invokes the CenterLine GNU debugger; see 'Distribution' on page iii for information about acquiring the source for this tool.

Once you are in process debugging mode, you can invoke the debugger using the **debug** command:

```
(pdm) 1 -> debug my.a.out
```

See the **debug** entry on page 113 for more information.

Within the ObjectCenter environment

If you are already in an ObjectCenter session using the Motif or OPEN LOOK version, you can switch to process debugging mode by selecting the **Restart Session** menu choice on the ObjectCenter pulldown menu. A dialog box allows you to restart the environment in either component debugging mode or process debugging mode.

Whenever you switch to process debugging mode from within the ObjectCenter environment, ObjectCenter initializes an ObjectCenter session using the standard startup file (**.pdminit**); it does not transfer any information to the new ObjectCenter session from the previous one. For instance, you lose all loaded files, linked libraries, and so on.

As previously mentioned, once you are in process debugging mode, you can invoke the debugger using the **debug** command:

```
(pdm) 1 -> debug my.a.out
```

NOTE If you are using Ascii ObjectCenter and you wish to switch to process debugging mode, you must start a new session from outside the environment.

Using pdm vs. cdm

You can use most ObjectCenter commands the same way, whether or not you are in process debugging mode, but there are a few differences.

NOTE When you are in process debugging mode, you cannot use the Project Browser, the Class Examiner, the Inheritance Browser, or the Cross-Reference Browser. Also, in pdm, ObjectCenter does not support most options, so the Options Browser is not available.

Commands

See Table 23 for a list of ObjectCenter commands supported by process debugging mode along with a description of any differences between the way each command works in component or process debugging mode**.** The shaded areas of the table indicate commands that are not available in process debugging mode. See the reference page for each command for more details about the command.

Table 23 Differences in ObjectCenter Commands by Mode

ObjectCenter Command	Check (✓) If Available in pdm	Differences between Component Mode (cdm) and Process Mode (pdm) Use of Command
action		Not implemented in pdm. Use **when**.
alias	✓	You can use **alias** *[name[text]]* in pdm, but you cannot use the following form in pdm: **alias** *name text alias_args*.
assign	✓	No difference.
attach	✓	Available in pdm only.
browse_base		Not implemented in pdm.
browse_class		Not implemented in pdm.
browse_data_members		Not implemented in pdm.
browse_derived		Not implemented in pdm.
browse_friends		Not implemented in pdm.
browse_member_functions		Not implemented in pdm.

Table 23 Differences in ObjectCenter Commands by Mode (Continued)

ObjectCenter Command	Check (✓) If Available in pdm	Differences between Component Mode (cdm) and Process Mode (pdm) Use of Command
build	✓	In pdm, ObjectCenter reloads **a.out** if **a.out** is newer than the current **a.out**.
catch	✓	No difference.
cd	✓	No difference.
classinfo		Not implemented in pdm.
config_c_parser		Does not apply to pdm.
construct		Not implemented in pdm.
cont	✓	The pdm, but not cdm, version of the **cont** command supports the following syntax: **cont at** *line* Continue at location specified by *line* **cont at** *line* **sig** *signum* Continue with last signal encountered cont **sig** *signum* Continue with signal specified by *signum* cont **skip** *count* Continue, ignoring breakpoint for *count* iterations The pdm version of the **cont** command does *not* support the following syntax: **cont** *continuation_value*
contents	✓	The pdm version of the **contents** command returns the pathname of the **a.out** file currently loaded. The **contents** *filename* variation may return only a partial list of objects declared or defined in *filename*.
cxxmode		Not implemented in pdm.

Table 23 Differences in ObjectCenter Commands by Mode (Continued)

ObjectCenter Command	Check (✓) If Available in pdm	Differences between Component Mode (cdm) and Process Mode (pdm) Use of Command
debug	✓	Available in pdm only.
delete	✓	Available in pdm: **delete** *n* **delete all** Not available in pdm: **delete** **delete** *file:line*
destruct		Not implemented in pdm**.**
detach	✓	Available in pdm only.
display	✓	No difference.
down	✓	No difference.
dump	✓	No difference.
edit	✓	No difference.
email	✓	No difference.
english		Not implemented in pdm.
expand		Not implemented in pdm**.**
fg		Not implemented in pdm.
file	✓	No difference.
gdb	✓	Available in pdm only.
gdb_mode	✓	Available in pdm only.
help	✓	No difference.
history	✓	No difference.

Table 23 Differences in ObjectCenter Commands by Mode (Continued)

ObjectCenter Command	Check (✓) If Available in pdm	Differences between Component Mode (cdm) and Process Mode (pdm) Use of Command
ignore	✓	No difference.
info		Not implemented in pdm.
instrument		Does not apply to pdm.
keybind		Not implemented in pdm.
link		Does not apply to pdm.
list	✓	No difference.
list_classes		Not implemented in pdm**.**
listi	✓	Available in pdm only.
load		Does not apply to pdm. Use **debug**.
make	✓	Using the ObjectCenter syntax in makefiles has no effect in pdm.
man	✓	No difference.
next	✓	No difference.
nexti	✓	Available in pdm only.
print	✓	ObjectCenter uses different formats in cdm and pdm for displaying the value of an expression or variable.
printenv	✓	No difference.
printopt	✓	No difference.
proto		Does not apply to pdm.
quit	✓	In pdm you do not have the choice of saving to a project file.
reinit		Does not apply to pdm.

Table 23 Differences in ObjectCenter Commands by Mode (Continued)

ObjectCenter Command	Check (✓) If Available in pdm	Differences between Component Mode (cdm) and Process Mode (pdm) Use of Command
rename		Does not apply to pdm.
rerun	✓	No difference.
reset	✓	No difference.
run	✓	No difference.
save		Does not apply to pdm.
set	✓	No difference.
setenv	✓	No difference.
setopt	✓	No difference.
sh	✓	No difference.
shell	✓	No difference.
source	✓	No difference.
start		Does not apply to pdm.
status	✓	No difference.
step	✓	No difference.
stepi	✓	Available in pdm only.
stepout	✓	No difference.
stop	✓	See the ObjectCenter *Platform Guide* for information about setting breakpoints in shared libraries while in pdm.
stopi	✓	Available in pdm only.
suppress		Does not apply to pdm.
suspend		Does not apply to pdm.

Table 23 Differences in ObjectCenter Commands by Mode (Continued)

ObjectCenter Command	Check (✓) If Available in pdm	Differences between Component Mode (cdm) and Process Mode (pdm) Use of Command
swap		Does not apply to pdm.
touch		Does not apply to pdm.
trace		Not implemented in pdm.
unalias	✓	No difference.
uninstrument		Does not apply to pdm.
unload		Does not apply to pdm.
unres		Does not apply to pdm.
unsetenv	✓	No difference.
unsetopt	✓	No difference.
unsuppress		Does not apply to pdm.
up	✓	No difference.
use	✓	No difference.
whatis	✓	No difference.
when	✓	Available in pdm only.
where	✓	No difference.
whereami	✓	No difference.
whereis	✓	No difference.
xref		Not implemented in pdm.

Using gdb in process debugging mode

As a convenience, ObjectCenter allows you to use **gdb** commands in the Workspace when you are in process debugging mode; **gdb** is the GNU source-level debugger provided by the Free Software Foundation.

If you wish to use **gdb**, you can do any of the following:

- Invoke ObjectCenter at the shell prompt with the **-gdb** command-line switch in addition to the **-pdm** switch:

  ```
  $ objectcenter -pdm arg1 ... -gdb argn ...
  ```

  If you do so, all command-line arguments after the **-gdb** are taken to be **gdb** command-line switches, and any switches before the **-gdb** are taken to be pdm switches.

- Issue the **gdb** command in the Workspace while you are in process debugging mode; the **gdb** command takes as its argument any **gdb** command:

  ```
  (pdm) 1 -> gdb break 20
  ```

- Issue the **gdb_mode** command in the ObjectCenter Workspace while you are in process debugging mode:

  ```
  (pdm) 1 -> gdb_mode
  ```

Once you are in **gdb** mode, you can use only the **gdb** command set. You can get back to process debugging mode by typing the following command:

```
(gdb) pdm
```

NOTE Although we provide access to native **gdb** commands as a convenience, we do not provide any technical support for **gdb.**

See Also

attach, **commands**, **debug**, **detach**

performance

performance trade-offs and enhancements

ObjectCenter provides two debugging modes: process debugging mode and component debugging mode. There are performance trade-offs between the two modes that you need to consider before starting a debugging session. There are also several ways you can load your code in a cdm session that affect performance.

Process debugging mode is especially useful for finding bugs in existing code, rather than for debugging new code as you develop it. It offers the fastest startup and execution time and reasonably good debugging facilities.

In component debugging mode you can load source or object code, you can load your object code with or without debugging information, and you can "instrument" your object code to add run-time error-checking capabilities (see the **instrument** entry on page 162). Which combination of these you choose depends on what your goals are and how large your project is.

This entry contains the following sections to help you make trade-off decisions and take advantage of other features you can use to enhance performance:

- Performance factors for source and object components
- Enhancing performance for large projects
- Additional performance enhancements

For a more detailed discussion of the effects of debugging techniques on performance, see the **debugging** entry on page 116.

Performance factors for source and object components

As described above, when you are in component debugging mode, you can load several different combinations of source and object code. Table 24 summarizes the performance characteristics of each source or object format. The number 1 represents the fastest speed and the least memory consumption.

Table 24 Performance Characteristics of Source and Object Code

	Setup speed	Memory use	Execution speed
Source code	5	5	3
Instrumented object code with debugging information	4	4	2
Instrumented object code without debugging information	3	2	2
Regular object code with debugging information	2	3	1
Regular object code without debugging information	1	1	1

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

As Table 24 shows, regular object code without debugging information offers fastest speed of setup and execution and least memory use, while source code takes longest to set up and execute and requires the most memory. Table 25 shows that source code offers the best error checking and debugging capabilities while regular object code without debugging information offers the most limited. Specific limitations are shown after the table.

Table 25 Error-Checking and Debugging Capabilities in Source and Object Code

1 = full 2 = some restrictions 3 = limited 4 = minimal	**Load-time error checking**	**Run-time error chjecking**	**Debugging**	**Code visualization**
Source code	1	1	1	1
Instrumented object code with debugging information	4	2	2	2
Instrumented object code without debugging information	None	3	3	3
Regular object code with debugging information	4	4	2	2
Regular object code without debugging information	None	4	3	3

Some specific restrictions are as follows:

Load-time error checking

The only load-time error-checking of object code with debugging information performed is consistency checks of declarations and definitions across modules.

Run-time error checking

Run-time error checking of uninstrumented object code only covers certain standard library functions such as **malloc()** and **strcpy()**.

Standard debugging actions

The following standard debugging actions are not available for any form of object code: tracing, the **stepout** command, and some forms of the **action** command. In addition, you can set breakpoints on functions in object code without debugging info, but you cannot perform any other debugging actions.

Code visualization

The only code visualization available for object code without debugging information is cross-referencing of functions and definitions for global symbols.

The quality of type information available for object code with debugging information depends on how complete the debugging information supplied by your compiler is. Code visualization for object code with debugging information has the following restrictions for examining some variables:

- No class or function template information
- No information on type const
- No information on protection level for class members
- References are treated as pointers
- Class browsing is only available on the class hierarchy

To work around these restrictions, load a header file with the appropriate declarations, or swap one of the object files to source form.

Enhancing performance for large projects

Table 26 summarizes the advantages of the techniques that are likely to be most helpful with large projects. However, as already discussed, increased performance in loading, executing, and memory conservation are all factors that need to be balanced against debugging capabilities available.

Table 26 Performance Gains for Large Projects

	Performance enhancement gained		
Technique to use	**Setup speed**	**Speed of execution**	**Memory conservation**
Consolidate object files	✔		✔
Load object, not source	✔	✔	✔
Do not load debugging information	✔	✔	✔
Use regular object code, not instrumented code	✔	✔	✔
Set the **save_memory** option			✔

Additional performance considerations

In this section we list some additional ways to enhance performance.

Load object code without debugging information

Object code with debugging information requires much more memory and loads more slowly, so if memory or speed are issues, load object code without debugging information. You can do either of the following:

- Load an object file that was not compiled with the **-g** switch.
- Specify the **-G** switch with the **load** command to load the object files even faster. The **-G** switch makes the loader skip the **-g** debugging information in the object file. However, the only debugging you will be able to do on the resulting object code is to set breakpoints and actions on a particular function.

Consolidate object files

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

The loading time for the consolidated object file will be much faster than the total time for the separate smaller files.

To do this, invoke the UNIX linker as follows:

```
% ld -r -o all.o file1.o file2.o file3.o ...
```

This creates one large object file called **all.o** from the several smaller files named **file1.o, file2.o**, and so on. Loading **all.o** into ObjectCenter is faster than loading the individual object files.

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

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

Set the save_memory option

You can conserve memory by setting the **save_memory** option.

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

You need to set the **save_memory** option before you load files. Setting this option reduces run-time violation checking capabilities. For more information on the **save_memory** option, see the **options** entry on page 229 for more information about options.

Set instrument_space option to 0

The **instrument_space** option specifies the amount of space reserved for instrumentation of object files. The default value of this option is **2**, which corresponds to an amount of space approximately half the size of the text space of your application. Setting this option to **0** saves memory, but removes the ability to instrument the object file.

Skip header files when compiling object code

If you load object files that require compilation (they do not exist or are out-of-date with their source or header files), you can decrease the compilation time with precompiled header files, using ObjectCenter's facility for avoiding recompilation of common header files.This facility saves and reuses an image of the compiled code for header files that are used in common by modules in your program. To enable header file skipping, use the **+k** switch to **CC**. For more information, see the **precompiled header files** entry on page 271.

Use demand-driven code generation

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

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

To turn demand-driven code generation on or off, use the **-dd=on** or **-dd=off** switch, or select Enable Demand-Driven Generation in the Properties window for an individual file or for your project. The Properties windows are available from the Project Browser. For more information, see the **demand-driven code generation** entry on page 129.

Setting the dimButtonsWhen DebuggerBusy resource

The **dimButtonsWhenDebuggerBusy** resource can improve performance by reducing the X11 server traffic that results from dimming the control buttons in the GUI. This resource is especially valuable when running ObjectCenter with slow X servers or low-speed connections such as X over serial lines.

The resource enables you to specify the length of time that the debugger must be busy before the control buttons on the GUI dim. By default, the buttons on the GUI dim when the debugger has been busy for 1.15 seconds:

```
ObjectCenter*dimButtonsWhenDebuggerBusy: 1.15
```

You can change the value of this resource in the site-wide application defaults file for ObjectCenter, or in your local **.Xdefaults** file. The value can be:

- The string Always if you want the buttons to dim as soon as the debugger is busy.
- The string Never if you never want the buttons to dim.
- Any positive floating-point number, to indicate the number of seconds you want to elapse before the buttons start dimming.

porting

See the **cc and other C compilers** entry on page 40 and the **CC** entry on page 44.

precompiled header files

using ObjectCenter's facility for avoiding recompilation of common header files.

TIP: Using +k with CC to reduce compilation time for certain kinds of large programs

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

You can also use **+k** with the **load** command in the ObjectCenter environment; doing so tells ObjectCenter to use a precompiled header when loading an object file that requires compilation.

Overview

ObjectCenter's version of the AT&T C++ Language system provides a facility to keep track of header files that have been compiled and avoid recompiling them unnecessarily on subsequent compilations of the same program.

Using +k switch with CC command

You can invoke ObjectCenter's version of the C++ language system outside the ObjectCenter environment by using the following command-line:

CC *[switch] ... filename ...*

See the for more information about all the switches and filenames used with **CC**; in this entry we focus on the following switches that relate to precompiled header files:

-hdrepos =*dir_name* Use *dir_name* as a repository, and look in *dir_name* for the *filename* specified with **+k=***filename*.

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

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

Usage

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

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

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

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

For example:

```
# this is my precompiled header information file
<stdio.h> <string.h> “my_hdr1.h” -hdrepos ./my_repos
<stdio.h> <string.h> “my_hdr2.h” -hdrepos ./my_repos
<stdio.h>
```

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

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

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

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

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

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

- Changes to the included files
- If you change the arguments to any CC switch (such as **-D**, **-U**, **-I**, or **-dd={on|off}**) that affects the contents of a source file, it causes the precompiled header mechanism to treat any files in the repository as outdated. As a result, ObjectCenter recompiles

and saves the state of the newly compiled files rather than restoring an earlier state from the repository. Switches passed on to the C compiler or linker do not have this effect.

- Adding a comment causes the output of the translator to vary, so it causes a recompilation.
- When the time of the machine on which CC is executing is later than the time of the machine that the repository is written to, CC issues this warning:

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

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

Usage

Suppose you specify a precompiled header information file as follows:

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

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

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

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

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

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

In this example, the compiling system saves and restores the initial compilation results for **stdio.h** only. This is because there is no match in the precompiled header information file for any sequence of files except a sequence containing only the first one, **stdio.h.**

ObjectCenter saves the initial compilation results for **stdio.h** and restores them as needed for later compilations; the **string.h** and **my_hdr3.h** header files would be recompiled during every recompilation of this source file.

Restrictions

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

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

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

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

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

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

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

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

preprocessed code

ObjectCenter helps you debug preprocessed code by allowing you to examine the input to a preprocessor rather than just the output from it; the input is typically much easier to read than the output. ObjectCenter uses **#line** directives in the preprocessed code to map the preprocessed code to the unpreprocessed code that you wrote. See Figure 4 for a conceptual illustration.

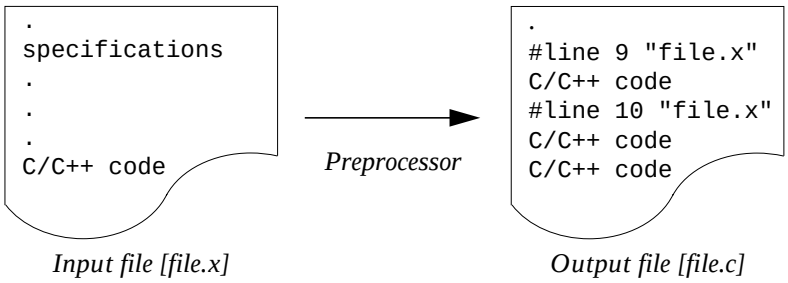

Figure 4 Preprocessor Input and Output in ObjectCenter

Supported preprocessors take an input file that consists of specifications and / or C++ or C code and generate a pure C++ or C file from it. The file contains **#line** directives referencing the input file.

In other words, you can use ObjectCenter to work with files generated by **yacc**, certain SQL preprocessors, and some preprocessors supporting parameterized types.

We discuss the following topics related to preprocessed code:

- Overview
- A sample program
- Loading the preprocessor output file
- Working with the preprocessed code
- Using commands
- Modifying the preprocessor input file

- Having ObjectCenter ignore **#line** directives
- Using preprocessors that do not generate **#line** directives

ObjectCenter uses the **#line** directives to associate lines in the generated C file with lines in the file that you wrote.

Overview

To work with preprocessor output, load a preprocessed C file that has **#line** directives in it along with the unpreprocessed input file used to create it. Then, for example, you can set breakpoints in the input file and run the program. ObjectCenter stops accordingly and displays the line in the input file. Similarly, you can make a change to an input file, issue **build**, and ObjectCenter creates a new output file and loads it.

NOTE You can also work directly with the output file if you want, just as with any other C file. If you want ObjectCenter to ignore **#line** directives, see the **ignore_sharp_lines** option described on page 242.

The rest of this description uses a C **yacc** program to illustrate the process. The techniques are similar with any C preprocessor that generates **#line** directives.

A sample program

The sample **yacc** program is a primitive calculator that uses a lexical analyzer generated by **lex**. It consists of two files**: calc.y** and **calc.l**.

calc.y:

```
%token INT
%token ADD
%token SUB
%token MULT
%%

lines:
| lines line
{
printf(" Total is %d\n", $2);
printf(" Try another, or 'quit' to leave.\n");
}
;
```

```
line:expr '\n'
{$$ = $1;}
;

expr:INT  {$$ = $1;}
|  ADD INT INT  {$$ = $2 + $3;}
|  SUB INT INT  {$$ = $2 - $3;}
|  MULT INT INT  {$$ = $2 * $3;}
;

%%
#include "lex.yy.c"
main()
{
printf("To add two numbers, type 'add num1 num2'\n");
printf("To subtract two numbers, type 'sub num1
num2'\n");
printf("To multiply two numbers, type 'mult num1
num2'\n");
yyparse();
}
```

calc.l:

```
%%
[0-9]+{
yylval = atoi(yytext);
return (INT);
}

\nreturn ('\n');

add return (ADD);
sub return (SUB);
mult return (MULT);

quit |
q return (0);

. ;
```

Because **yacc** generates **#line** directives in its generated C code, we will be able to work with **calc.y** in our ObjectCenter session. We will not have to look at **yacc**'s generated C code.

However, because some versions of **lex** do not generate **#line** directives in C code, we may not be able to work with the **lex** specification in **calc.l**.

NOTE Some variants of **lex** generate **#line** directives in output files. You can work with input files for such preprocessors using the procedures described below.

To compile this program outside of ObjectCenter, we enter the following:

```
% lex calc.l
```

which produces **lex.yy.c** as output, and

```
% yacc calc.y
```

which produces **y.tab.c** as output, and

```
% cc -o calc -g y.tab.c -ly -ll
```

which produces **calc** as an executable.

We will debug **calc.y**, from which **yacc** generates C code (**y.tab.c)**.

Loading the preprocessor output file

To start, you load the output file and any necessary libraries. In the case of our example, you need to issue the following command in the Workspace:

```
-> load -C y.tab.c -ly -ll
```

NOTE When you work with a file that has been run through a preprocessor, you may get many load-time and run-time warnings. Automatic code generators often produce code that has poor style by human standards. You can simply suppress the warnings and continue.

As ObjectCenter loads the output file, it uses the **#line** directives to relate the output file to any input files; in this case to **calc.y**. Because the two files are related through the **#line** directives in the loaded file, you can view and manipulate the code through the input file.

Loading object code

If you want the generated C code loaded in object form, make sure it has been compiled with debugging information; the **#line** directives that ObjectCenter requires are in the debugging information.

If the output file needs updating

In the following two situations, ObjectCenter does not immediately load the output file:

- If the output file is older than any input file referenced in a **#line** directive
- If the output file does not exist

If the output file is out-of-date

In the first situation, you issue **load** to load a file containing **#line** directives, and the file is older than any of the referenced input files. In this case, ObjectCenter determines that the output file needs to be updated, and it requires a makefile to do so. The makefile must have a target that specifies how to create the output file from the input file(s).

In the case of our **calc** example, the following target rule would provide the necessary information:

```
y.tab.c: calc.y
     yacc calc.y
```

This rule says to create **y.tab.c** by issuing the command **yacc calc.y**.

Similarly, if you are using an SQL preprocessor, you would enter in your **make** target the shell command you would issue to process the code containing the embedded SQL.

Here is what happens when you load an output file that is older than the input file, and there is a **make** target to build it:

```
-> sh touch calc.y                    "update" calc.y
-> load -C y.tab.c
Loading (C): y.tab.c
File 'calc.y' was modified after 'y.tab.c'
Executing: make y.tab.c
yacc calc.y
Reloading (C): y.tab.c
```

The **touch** shell command updates **calc.y.** Then, as ObjectCenter loads the output file **y.tab.c**, it determines through the **#line** directives that there is a dependency on **calc.y.** Next, since **y.tab.c** is older than **calc.y**, ObjectCenter recreates the C file by issuing the command **make y.tab.c.**

If the output file does not exist

If you ask to load a source output file that does not exist, ObjectCenter cannot do anything *unless* you have used the **create_file** option to specify how to create the file.

The **create_file** option takes a string in the following format:

@file1@command1@file2@command2 **...** *@filex@commandx*

Note that the string consists of **file–command** pairs. For example, if you try to load **file1**, which does not exist, ObjectCenter issues the shell command **command1** to create the file, then loads it.

So, to handle the situation where **y.tab.c** does not exist, you could specify the following:

```
-> setopt create_file @y.tab.c@yacc calc.y
```

This says to create **y.tab.c** and issue the shell command **yacc calc.y**. Notice the similarity to the specification in the **make** target rule.

Here is an illustration:

```
-> unsetopt create_file
-> load -C y.tab.c
Cannot open '/s3/bobh/calc/y.tab.c'.
-> setopt create_file @y.tab.c@yacc calc.y
-> load -C y.tab.c
Cannot open '/s3/bobh/calc/y.tab.c'.

Executing: yacc calc.y
Loading (C): y.tab.c
```

Working with the preprocessed code

Once you have loaded the output file either in source form or in object form with debugging information, you can work directly with the input file.

Setting breakpoints

You can set a breakpoint and actions on any line in your input file that is generated into C++ or C code. You don't need to find the line in the executable code corresponding to the line in the input file. For example, you can set a breakpoint in **calc.y** by listing **calc.y**, and then clicking with the mouse in the Source panel.

This means you can set a breakpoint and actions on any line in your input file that is generated into C code. You do not need to find the line in the executable code corresponding to the line in the input file.

For example, you can set a breakpoint in **calc.y** by listing **calc.y**, and then using the **stop** command:

```
-> stop at "calc.y":10
stop (1) set at "calc.y":10, yyparse()
```

ObjectCenter automatically handles the mapping of that line to the line in the "real" code; in this case, the "real" code is in **y.tab.c**. Now you can run the program, and ObjectCenter stops where you want.

NOTE You cannot set a breakpoint on a line in the input file that does not correspond to a line in the output file through a **#line** directive.

Stepping through code

You can also step through code as long as the code you are stepping through is defined in the input file. In other words, the executable statement in the output file must be mapped back to the input file through a **#line** directive.

If you step into code that was generated by a preprocessor — that is, an executable statement in an output file that does *not* have a **#line** mapping to an input file — ObjectCenter shows you the generated code.

Using commands

If you have loaded an output file with **#line** directives, ObjectCenter always references the input file when dealing with lines that are mapped to an input file. For example, the informational commands, such as **where, whereis**, and **whereami**, display the line number and filename of the input file, if appropriate.

For example, suppose you are stopped at the line where we previously set a breakpoint in the **calc** example. If you issue **where**, you would see this:

```
(break 1) 10 -> where
stop #1 set in
yyparse() at "calc.y":10
main() at "calc.y":32
centerline_run((char *) 0x1533f0 "")builtin function
```

Listing functions

Similarly, if you ask to list a function that is defined in an input file, ObjectCenter displays the input file.

Swapping to object form

You can swap the output file between source and object code using the **swap** command. Make sure that the output file is compiled and loaded with debugging information so the **#line** directives are accessible to ObjectCenter.

If the object file does not exist or is out-of-date, ObjectCenter issues **make** to build it.

Modifying the preprocessor input file

You can debug and modify the input file just as you can debug and modify any standard C files. If you make and save a change to the input file, issue **build** to update your project. ObjectCenter can determine through the **#line** directives that the output file needs to be updated.

```
-> /* make and save change to calc.y */
-> build
Executing: make y.tab.c
yacc calc.y
Reloading (C): y.tab.c
```

Keep in mind that ObjectCenter requires a specific target in a makefile to generate an output file. If there is no such target, ObjectCenter cannot update the output file and so it remains out-of-date.

NOTE Always use **build** (either in the Workspace or in the Graphical User Interface) to update your project when you are working with preprocessed files. Using **reload** will not work because the input file is not actually loaded.

Having ObjectCenter ignore #line directives

If you want to work directly with the output file, ignoring completely the input file, you can set the **ignore_sharp_lines** option:

```
-> setopt ignore_sharp_lines
```

With this option set, ObjectCenter simply ignores the **#line** directives and no longer maintains any relationship between an input file and an output file. You work with the generated output file the same way you work with other C++ or C files. Your debugging is restricted to the output file.

NOTE The **ignore_sharp_lines** option applies only to source code. Object code debugging always uses the information provided by the compiler.

Using preprocessors that do not generate #line directives

Some preprocessors, such as some versions of **lex**, do not generate **#line** directives in their output file.

If you want to work directly with input files for such preprocessors, you must create a C++ or C output file with the appropriate **#line** directives. You can do this by hand, or write a program, script, or **make** target that generates the C++ or C file from the input file. You might need to consult a C or C++ language reference manual for the semantics of **#line** directives.

After you have inserted the required directives, load the output file. ObjectCenter will use the **#line** directives to do the mapping.

print

prints the value of variables and expressions

cdm	pdm
✔	✔

Command syntax

print *expression*

print *variable*

Description

expression Evaluates the specified expression and displays the resulting value.

variable Displays the value of the specified variable.

ObjectCenter uses different formats in cdm and pdm for displaying the value of an expression or variable.

Options

The following ObjectCenter options affect the **print** command:

print_inherited Filters the display of inherited data members.

print_pointer Adds diagnostic information to pointer display.

print_runtime_type Specifies pointer display as run-time rather than compile-time types.

print_static Specifies the display of static data members.

print_string Uses the number specified as the number of characters to print.

show_inheritance Tells ObjectCenter to show full rather than truncated inheritance path.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **print** command to check the current value of variables and expressions. ObjectCenter prints their values in the Workspace.

```
-> print r
(struct Rectangle *) 0x8a98
```

The value of a variable or expression can also be displayed without the **print** command. This is accomplished by evaluating the variable or expression directly in the Workspace:

```
-> print 123+456
(long) 579
-> 123+456;
(long) 579
->
```

In component debugging mode (not in pdm), you can specify the location of a variable in one of four ways:

- *`file`function`variable*
- *`file`line_number`variable*
- *`file`variable*
- *function`variable*

See Also

assign, **display**, **dump**, **list**, **whatis**, **whereis**

printenv

displays the system environment

cdm	pdm
✔	✔

Command syntax

printenv

printenv *variable*

Description

<< *none* >>	Lists all currently defined environment variables.
variable	Displays the value of the specified environment variable, if that variable is currently defined.

Usage

Use the **printenv** command in conjunction with **setenv** and **unsetenv** to manipulate the variables in the program's system environment. The **printenv** command is similar to the shell command with the same name.

Warnings

The **printenv** command displays the default values of the environment variables, which are the values that your program inherits each time it starts. Therefore, if a program has added any environment variables, for instance with the **putenv()** library function, the changes will not be shown by the **printenv** command.

If **setenv** or **unsetenv** is called from a break level, they will alter the value of the global **environ** variable, but not the **envp** parameter passed to **main()**. (This problem also occurs with the **putenv()** function.)

Changing the **EDITOR** or **DISPLAY** shell variables with these commands will not affect which editor or display screen ObjectCenter uses.

See Also

setenv, **unsetenv**, **environment variables**

printopt

displays information on ObjectCenter options

cdm	pdm
✔	✔

Command syntax

printopt

printopt *option*

Description

<< *none* >>	Displays a list of all ObjectCenter options.
option	Displays the current value of the specified option and gives a short description of its function.

Usage

Use the **printopt** command to examine current settings for ObjectCenter options.

To retrieve the value of an option from within a function, use the ObjectCenter function **centerline_getopt()**. This function returns the current value of the option as a string. For more information about ObjectCenter functions, see the **built-in functions** entry on page 34.

See Also

built-in functions, **centerline_getopt()**, **setopt**, **unsetopt**

process debugging mode

supports debugging fully linked executable files and attaching to a process.

See the **pdm** entry on page 254.

properties

When you use the Motif or OPEN LOOK versions of ObjectCenter, you can use the Project Browser to specify the switches and options you want ObjectCenter to use when it loads a file.

These specifications are known as Properties, and you can set them in the Project-wide Properties window; select **Project Properties** in the **Project** pulldown menu on the Project Browser to access the Project-wide Properties window. You can also set Properties for individual files by selecting **Properties** in the Project Browser window.

For more information on the options and switches related to properties, see the **load** entry on page 185.

See Table 27 for a list of Properties and a brief description of each.

Table 27 Project Properties and Their Corresponding ObjectCenter Options

Project-Wide Property	Description	Corresponding Option
Program Name	Specifies the value of the first argument to **main()**; that is, **argv[0]**.	**program_name**
Search Path for Files	If the **swap_uses_path option** is set, specifies the order for searching directories when any of the following ObjectCenter commands are invoked: **cd**, **edit**, **list**, **load**, or **swap**.	**path**
Use Search Path When Swapping	Specifies whether to use the value of the **path** option when the **swap** command is invoked.	**swap_uses_path**
Load Flags	Specifies the default switches used if the **load** command is called without switches. Do not specify **-w** or **-G** switches here.	**load_flags** (exclusive of **-G** and **-w** switches**)**

Table 27 Project Properties and Their Corresponding ObjectCenter Options (Continued)

Project-Wide Property	Description	Corresponding Option
Assume ANSI C	Specifies whether to strictly conform to the ANSI C standard for preprocessing and function prototype conversion.	**ansi**
Ignore Warnings When Loading	Specifies whether to automatically suppress all load-time warnings.	**-w** switch for **load_flags**
Load Debugging Information	Specifies whether to load debugging information contained in object files compiled with the **-g** switch.	**-G** switch for **load_flags**
Instrument Object Files	Specifies whether to automatically instrument each object file when it is loaded.	**instrument_all**

proto

generates prototypes for C functions and writes them to a file

cdm	pdm
✔	

Command syntax

proto all

proto *file*

proto user

Description

all	Generates function prototypes for all functions currently defined and writes these prototypes out to a file. Defined functions include functions defined in the Workspace.
file	If the specified file is currently loaded, generates function prototypes for all the functions defined in the specified file and writes these prototypes out to a file.
user	Generates function prototypes for all functions defined in currently loaded files and writes these prototypes out to a file. Note that functions defined in the Workspace are not included.

Usage

Use the **proto** command to generate and list function prototypes for defined C functions. The prototypes that **proto** generates meet ANSI C standards. To provide ANSI C prototyping in your C source code, use an **#include** statement to include the prototype output file at the head of your program.

If you do not specify a file when issuing the **proto** command, **proto** prompts for the name of an output file. If the named file exists, you are asked whether you want to overwrite it, append to it, or choose a new filename.

Usage

The following example shows how **proto** can be used to create a prototype file from a group of C source files. If the files **foo.c, bar.c**, and **bam.c** are the source files for a common library **mylib.a**, a prototype file for **mylib.a** can be generated for the **.c** files as follows:

```
-> load foo.c bar.c bam.c
Loading (C): foo.c
Loading (C): bar.c
Loading (C): bam.c
 -> proto user
Writing prototypes to a file.
Output file name? mylib.proto
 ->
```

Assume that **foo.c** contains the following definition:

```
int fl_int() {return 3;}
```

In a new ObjectCenter session, you can load the prototype file for **mylib** to provide type checking for calls to functions in **mylib.a**:

```
-> load mylib.proto
Loading (C): mylib.proto
 -> fl_int(3, 4);
Warning #65: Calling function 'fl_int' with too many
parameters. Passing 2, expecting 0.
Defined/declared in "mylib.proto":38
```

Restrictions

Due to bugs in some C compilers, types in object file symbol tables are often wrong. To get the most reliable results from the **proto** command, apply **proto** only to modules loaded in source form.

See Also

load, **unload**

quit

quits ObjectCenter

cdm	pdm
✔	✔

Command syntax

quit

quit force

quit project

quit project *file*

Description

<< *none* >>	Exits ObjectCenter and returns you to the shell. In component debugging mode, prompts you first to save the state of your session in a project file.
force	Exits ObjectCenter and returns you to the shell. You are *not* given an opportunity to save the state of your session before exiting.
project	Exits ObjectCenter saving your project in a file named **ocenter.proj**. (cdm only)
project *file*	Exits ObjectCenter saving your project in a file named *file*. (cdm only)

Usage

Use the **quit** command to exit ObjectCenter and return to the shell.

Before exiting, ObjectCenter notifies you if there are any active editing jobs. If you requested a logfile on the command line when starting ObjectCenter, the name of the logfile is displayed as part of the exit message.

Restrictions

In pdm, you do not have the choice of saving to a project file.

See Also

save, **suspend**

reinit

initializes all global variables

cdm	pdm
✔	

Command syntax

reinit

reinit *variable*

Description

<< *none* >>	Sets all global variables to their initial values, frees allocated memory, closes open files, and sets all signal handlers to their initial values. Freed memory is not returned to the system; it is marked as free within ObjectCenter, to be reused the next time you run your program.
variable	Sets the specified variable to its initial value.

Usage

Use the **reinit** command to reinitialize either a specific variable or your entire program. By using the **reinit** command, you either set a particular variable to its initial value or set all global variables to their initial values and also free allocated memory, close open files, and set all signal handlers to their initial values.

If **reinit** is called from a break level, errors may be introduced when global variables are reinitialized to their initial values. In addition, **reinit** will free allocated memory that might still be used.

To remove definitions of static structures from the Workspace, use **unload workspace** instead of **reinit**.

See Also

construct, **rerun**, **run**, **start**

rename

renames an ObjectCenter function

cdm	pdm
✔	

Command syntax

rename

rename *old_name new_name*

Description

<< *none* >>	Lists all functions that have been renamed.
old_name new_name	Changes the name of an ObjectCenter function (*old_name*) to the name given as the second argument (*new_name*).

Usage

Use the **rename** command to change the name of an ObjectCenter function to prevent name conflicts with the variables or functions defined by the user's program.

You should rename the conflicting function name before any files are loaded to avoid accidental use of the ObjectCenter function to resolve external references in user code. The **rename** command will not rename a function if it has already been used to resolve a reference within your program.

Saving a project file, however, does not save any renaming. To permanently record renamed ObjectCenter functions, place the **rename** command with *old_name new_name* arguments in your **.ocenterinit** file for each renaming.

Restrictions

The **rename** command has no effect in C++ mode. To use this command, use the **cmode** command to change modes.

See Also

alias, **keybind**, **xref**

rerun

executes **main()** with new arguments

cdm	pdm
✔	✔

Command syntax

rerun

rerun *argument ...*

Description

<< none >> Clears any old command-line arguments, initializes all variables, and then executes **main().**

argument ... Clears any old command-line arguments, initializes all variables, processes the new command-line arguments (*argument ...*), and then executes **main()**.

If you issue a **rerun** command while you are at a breakpoint in **cdm**, ObjectCenter restarts and informs you that it is resetting the break level; instead, in pdm, ObjectCenter prompts you first before resetting the break level.

Options

The following ObjectCenter options affect the **rerun** command:

batch_run Specifies method for handling run-time violations.

lint_run Indicates the severity of warnings issued by ObjectCenter during execution.

program_name Specifies value of the first argument, **argv[0]**, to **main()**.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **rerun** command to execute **main()** with new arguments.

Arguments must be delimited by spaces. To include spaces in an argument string, precede each space with a backslash (\) character. Calling a program with **rerun** produces the same results as calling an executable program from the shell.

Restrictions

In the Workspace, to pass an argument with a space in it to **main(),** you must escape it with a backslash. Enclosing the argument in quotation marks, which works in a UNIX shell, does not work in the Workspace. For example, to call **main()** with two arguments, the first one containing the string **first arg**, and the second argument containing the number **3**, call **rerun** as follows:

```
-> rerun first\ arg 3
```

In contrast, if you are using the Motif or OPEN LOOK versions of ObjectCenter, you can use double quotes just as you do in a UNIX shell when you supply arguments for the **Run** dialog box.

See Also

reinit, **run**, **start**

reset

returns to a previous break level

cdm	pdm
✔	✔

Command syntax

reset

reset *number*

reset -*number*

Description

<< *none* >>	Returns execution to the top level of the Workspace.
number	Returns execution to the break level specified by *number*. (cdm only)
-number	Returns execution to the break level specified by subtracting *number* from the current break level**.** (cdm only)

Usage

Use the **reset** command to return to a previous break level without continuing execution from the current break level.

In process debugging mode, when you issue the **reset** command the executable is killed and its resources are freed.

Example

For example, to return execution from break level 5 to break level 3:

```
(break 5) 80 -> reset -2
Resetting to break level #2.
(break 3) 81 ->
```

See Also

cont, **stop**, **where**, **whereami**

revision control system support

ObjectCenter provides some predefined automatic revision control system commands. See the **X resources** entry on page 436.

run

executes **main()** with arguments

cdm	pdm
✔	✔

Command syntax

run

run *argument* **...**

Description

<< *none* >>	Initializes all variables, processes any command-line arguments from the previous call to **run** or **rerun**, and then executes **main()**.
argument ...	Clears any old command-line arguments, initializes all variables, processes the new command-line arguments (*argument ...*), and then executes **main()**.

Options

The following ObjectCenter options affect the **run** command:

batch_run	Specifies method for handling run-time violations.
lint_run	Indicates the severity of warnings issued by ObjectCenter during execution.
program_name	Specifies value of the first argument, **argv[0]**, to **main()**.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **run** command to execute **main()** after initializing all variables and processing any command-line arguments.

If you issue a **run** command while you are at a breakpoint, ObjectCenter restarts and informs you that it is resetting the break level.

When you run your program in ObjectCenter, it opens a separate Run Window for output and return control to the shell in which you invoked ObjectCenter. A CenterLine program called **clxterm** creates this Run Window, which is a standard version of **xterm**, the X11 terminal emulator.

To avoid creating the separate Run Window and avoid returning control to the shell, use the **-no_run_window** switch to the **objectcenter** command. With this switch, the program's input and output goes to the shell in which you invoked ObjectCenter. Using the **-no_run_window** switch means you are unable to interrupt ObjectCenter and unable to place it in the background. This option is intended for debugging applications that need specific terminal support rather than a generic terminal such as xterm.

To create a separate Run Window and avoid returning immediate control to the shell, use the **-no_fork** switch to the **objectcenter** command. With **-no_fork**, control returns when you enter the suspend character (usually **^Z**) in the shell or exit ObjectCenter. After you type the suspend character in the shell, you must type **bg** to enable your program to perform output again to the Run Window. Without **-no_fork**, the shell prompt comes back immediately.

How ObjectCenter interprets the run command

Both **run** and **rerun** construct arguments for **main()** from the command line. If **run** is called without any arguments, it uses the command-line arguments from the most recent call to either **run** or **rerun**. If **rerun** is called without any arguments, it calls **main()** without any arguments.

Not initializing variables

Using **run** or **rerun** to execute **main()** forces all global variables to be initialized. Before executing the program, both commands call **reinit**, which initializes all global variables. This may present a problem if you want to test special cases by setting variables to specific values prior to execution.

To avoid initializing global variables when executing **main()**, use the **start** command. The **start** command performs all the functions of **run** without calling **reinit**.

In the following example, the variable **seed** is set to a special value before **main()** is executed. To avoid having the value of **seed** reset to **0**, the **start** command is used instead of **run**.

```
-> whatis seed
extern int seed; /* initialized */
-> seed;
(int 7)
-> reinit
-> seed;
(int) 0
-> seed = 7;
(int) 7
-> start
Executing: a.out
```

Notice that **reinit** is called before **start**. It is important that **reinit** be called between calls to **start** to ensure that input/output buffers and other library data structures are initialized to their correct values.

Setting argv[0]

When a program is run from a shell, the value assigned to the variable **argv[0]** is the name of the program. The value of **argv[0]** in the example below is **echo**.

```
% echo abc.C xyz.C
```

In ObjectCenter, the value of **argv[0]** is set by the **program_name** option, for which the default value is **a.out**. You can change the value of **argv[0]** by changing the **program_name** option:

```
-> load echo.c
Loading: echo.c
-> setopt program_name echo
-> run this is a test
Executing: echo this is a test
this is a test
Program exiting with return status = 0
```

Shell used to process arguments

Any arguments that you supply with the **run** command are first passed to a shell, which expands wildcard characters, substitutes variables, and redirects I/O, and then passed to **main()**. The value of the SHELL environment variable, as outlined in Table 28, specifies the shell to be used for processing these arguments.

Table 28 Shells Used in Process Debugging Mode (pdm) with the **run** Command

Value of SHELL Environment Variable	What ObjectCenter Does
No SHELL environment variable	Uses **/bin/sh**
/bin/tcsh	Uses **/bin/csh** instead of **/bin/tcsh**[a]
/bin/csh	Uses **/bin/csh** with the **-f** flag[b]
All other values not listed	Invokes shell with the **-c** option.

a. This avoids a problem with **tcsh,** where the first file descriptor that the user program gets is 6 instead of 3.

b. This keeps the shell from reading your startup file and improves speed.

Passing arguments containing spaces

In the Workspace, in order to pass an argument with a space in it to **main(),** you must precede the space with a backslash. Enclosing the argument in quotation marks, which works in a UNIX shell, does not work in ObjectCenter.

For example, to call **main()** with two arguments, the first one containing the string **first arg**, and the second argument containing the number **3**, call **run** as follows:

```
-> run first\ arg 3
```

In contrast, if you are using the Motif or OPEN LOOK versions of ObjectCenter, you can use quotation marks just as you do in a UNIX shell when you supply arguments for the **Run** dialog box.

Running inside or outside of ObjectCenter

If you want your program to know at run time whether it is running in ObjectCenter, you can use the built-in function **centerline_true()**.

See Also

reinit, **rerun**, **start**

save

saves the current session in a project file

cdm	pdm
✔	

Command syntax

save

save *file*

save project

save project *file*

Description

<< *none* >>	Saves a project file with the default name **ocenter.proj.**
file	Saves a project file with the specified filename.
project	Same as << *none* >>. Saves a project file with the default name **ocenter.proj.**
project *file*	Same as *file*. Saves a project file with the specified filename.

Usage

Use the **save** command to save the current session of ObjectCenter so that it can be restored later.

Project files

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

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

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

NOTE A project file does not save variables or functions defined in the Workspace.

Loading project files

To load a saved project file, use the **load** command and supply the name of the file. You can also include the name of a project file on the command line when starting ObjectCenter.

Restrictions

Information about open files is not saved with the session. Thus, saving a session while files are open may create problems when the session is reloaded, unless the exact same files have been opened again. Also, open files that are not open in the reloaded session might be closed improperly and data could be lost.

The state of the terminal is not saved in a project file.

See Also

load

set

assigns a value to a variable

cdm	pdm
✔	✔

Command syntax **set** *variable* = *expression*

Description

variable **=** *expression* Evaluates *expression* and assigns its value to *variable*.

Usage

Use the **set** command to assign a value to a variable.

The specified variable can be a variable defined in either the program or the Workspace.

A variable can also be assigned a value without the **assign** or **set** commands, simply by evaluating an assignment expression in the Workspace, as follows.

```
-> int i;
-> set i = 2
(int) 2
-> i = 5;
(int) 5
```

See Also **assign**

setenv

adds a variable to the system environment

cdm	pdm
✔	✔

Command syntax

setenv

setenv *variable*

setenv *variable value*

Description

<< *none* >>	Lists all defined environment variables and gives their current values. This is equivalent to calling **printenv** without an argument.
variable	Defines *variable* and sets its value to the empty string. If the specified variable already exists, its value is reset to the empty string.
variable value	Defines *variable* and sets it to the value specified by *value*. If *variable* already exists, its value is reset to *value*.

Usage

Use the **setenv** command to manipulate the variables in the program's system environment. The **setenv** command is analogous to the shell command of the same name.

These commands affect only your program's environment variables. They do not affect the environment variables used by ObjectCenter to control its own operations.

The environment is an array of strings that is made available to the program through the global **environ** variable and the **envp** parameter, which is passed as the third argument to the **main()** function. By convention, each string has the format **name=value**, where the **value** part is optional.

You can use ObjectCenter's ability to expand environment variables and options (described in more detail in the Workspace entry) to add a string to an existing environment variable. In the following example, we add **/usr/shared/lib** to the existing LD_LIBRARY_PATH variable:

```
-> printenv LD_LIBRARY_PATH
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib
-> setenv LD_LIBRARY_PATH #$LD_LIBRARY_PATH:/usr/shared/lib
-> printenv LD_LIBRARY_PATH
LD_LIBRARY_PATH=/usr/lib:/usr/local/lib:/usr/shared/lib
```

Warnings

Be careful when checking the current values for environment variables. The **printenv** and **setenv** commands, when issued with no argument, display the default values of the environment variables, which are the values that your program will inherit each time it starts.

If **setenv** or **unsetenv** are called from a break level, they will alter the value of the global **environ** variable, but not the **envp** parameter passed to **main()**. This problem also occurs with the **putenv()** function.

Changing the **EDITOR** or **DISPLAY** shell variables with these commands will not affect which editor or display screen ObjectCenter uses.

See Also

environment variables, **printenv**, **setopt**, **unsetenv**

setopt

sets an ObjectCenter option

cdm	pdm
✔	✔

Command syntax

setopt

setopt *option*

setopt *option value*

Description

<< *none* >>	Displays all options and values that are currently set.
option	If the specified option is a Boolean, sets its value to TRUE. If the specified option takes a string, sets its value to the empty string. If the specified option takes an integer, sets its value to **1**.
option value	Sets *option* to the value specified by *value*.

Usage

Use the **setopt** command to examine and change ObjectCenter options. You can also use the Options Browser to examine and change options; see the *User's Guide* for more information.

You can use the standard C++ or C escape sequences (such as **\n** for newline) in option strings. To embed an Escape character, use **\e**.

You can use ObjectCenter's ability to expand environment variables and options (described in more detail in the Workspace entry) to add a string to an existing option. In the following example, we add **-L** and **-l** switches to the existing **load_flags** option setting to add **libnew.a**:

```
-> printopt load_flags
load_flags -DDEBUG -w
-> setopt load_flags #$load_flags -L/my_libs/libdir
-lnew
-> printopt load_flags
load_flags -DDEBUG -w -L/my_libs/libdir -lnew
```

To retrieve the value of an option from within a function, use the ObjectCenter function **centerline_getopt().** This function returns the current value of the option as a string. For more information about ObjectCenter functions, see the **built-in functions** entry on page 34.

See Also **built-in functions**, **centerline_getopt()**, **options**, **printopt**, **unsetopt**

sh

executes a Bourne subshell

cdm	pdm
✔	✔

Command syntax

sh

sh *argument* **...**

Description

<< *none* >>	Executes a Bourne subshell, setting no switches and passing no arguments.
argument ...	Executes a Bourne subshell, setting the **-c** switch and passing the specified arguments.

Usage

Use the **sh** command to execute a Bourne subshell. This can be used to execute UNIX commands from the Workspace:

```
-> sh rm my_file
```

See Also

shell

shared libraries

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

ObjectCenter supports shared libraries on all systems that provide them. Shared libraries typically link more quickly than static libraries in the ObjectCenter environment.

ObjectCenter does not read any debugging information on shared libraries in component debugging mode. Without debugging information on a file, you are unable to perform certain debugging activities, such as stepping through functions. For information about what debugging techniques are possible on code without debugging information, see the **debugging** entry on page 116.

In process debugging mode, ObjectCenter supports full source-level debugging of shared libraries that were compiled with **-g**.

Setting breakpoints in shared library functions

Keep in mind that when you are in component debugging mode, functions in shared libraries are not defined until your program references them. This means, for instance, that you cannot set breakpoints on a library function until you have linked or run your program. In the meantime, you may get messages indicating that the function is undefined. Here is an example:

```
Attaching: /usr/5lib/libc.sa.2.6
Attaching: /usr/5lib/libc.so.2.6
-> load ~/c_programs/sample.c
Loading: /s/users/jk/c_programs/sample.c
-> stop in printf
Cannot set stop or action on an undefined symbol:
'printf'.
-> link
Linking from '/usr/5lib/libc.sa.2.6' ... Linking
completed.
-> stop in printf
stop (1) set at "/usr/5lib/libc.so.2.6", function
printf().
```

See your *ObjectCenter Platform Guide* for information about setting breakpoints in shared library functions while you are in process debugging mode.

Restrictions in cross-referencing

Using **xref** to cross reference symbols in shared libraries is unreliable and changes depending on the execution state of your program. This is due to the way that symbols are linked from shared libraries. If you need accurate cross-reference information, load and link static libraries.

NOTE See your *ObjectCenter Platform Guide* for more information about the use of shared libraries on your particular platform.

---

shell

executes a subshell

cdm	pdm
✔	✔

Command syntax

shell

shell *argument ...*

Description

<< *none* >>	Executes the default shell set by the ObjectCenter **shell** option. In process debugging mode, executes the shell specified by the SHELL environment variable. Sets no switches and passes no arguments.
argument ...	Executes the default shell set by the ObjectCenter **shell** option. In process debugging mode, executes the shell specified by the SHELL environment variable. Sets the **-c** switch and passes *argument* to the shell. You can have more than one argument.

Usage

Use the **shell** command to execute the shell specified by the **shell** option. This can be used to execute UNIX commands in the Workspace.

Options

The following ObjectCenter option affects the **shell** command:

shell	Specifies the shell that is started by the **shell** or the **#!** commands.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

See Also

sh

source

reads ObjectCenter commands from a file

cdm	pdm
✔	✔

Command syntax **source** *file*

Description

file Reads ObjectCenter commands from *file*.

Usage

Use the **source** command to read ObjectCenter commands from a file. ObjectCenter uses **source** to read the system-wide startup file and either the **.ocenterinit** or **.pdminit** file in your home or current directory when you start ObjectCenter.

NOTE Any change in the break level causes the **source** command to terminate, therefore ObjectCenter will stop executing commands in the file if it encounters a breakpoint or a run-time violation. In addition, because the **step** command causes execution to go up a break level and then back to the original break level, ObjectCenter will stop reading commands from a source file after the first **step** command.

Example

The following example indicates how to use **source** with a file containing aliases:

```
% cat aliases
alias p print
alias s step
alias n next
% objectcenter
. . .
-> source aliases
-> p 123+456
(long) 579
->
```

See Also **load**

start

executes **main()** without initializing global variables

cdm	pdm
✔	

Command syntax

start

start *argument ...*

Description

<< *none* >>	Executes **main()** without initializing any global variables. When the program exits, **start** does not close any files. Uses any previous command-line arguments.
argument ...	Passes the specified command-line arguments and then executes **main()** without initializing any global variables. When the program exits, ObjectCenter does not automatically close any files.

Options

The following ObjectCenter options affect the **start** command:

batch_run	Specifies method for handling run-time violations.
lint_run	Indicates the severity of warnings issued by ObjectCenter during execution.
program_name	Specifies value of the first argument, **argv[0]**, to **main()**.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **start** command to execute **main()** without automatically initializing any variables and to exit without automatically closing any files on program exit. Use **start** for creating test situations that would be wiped out when **run** or **rerun** would initialize all variables.

Arguments must be delimited by spaces. To include spaces in an argument string, precede each space with a backslash (\) character.

Restrictions

In order to pass an argument with a space in it to **main(),** you must precede the space with a backslash. Enclosing the argument in quotation marks, which works in a UNIX shell, does not work in ObjectCenter. For example, to call **main()** with two arguments, the first one containing the string **first arg**, and the second argument containing the number **3**, call **run** as follows:

```
-> run first\ arg 3
```

See Also

construct, **reinit**, **rerun**, **run**

status

lists debugging items (actions, breakpoints, displayed items, and traces)

cdm	pdm
✔	✔

Command syntax

status

Description

<< *none* >> Lists all currently set debugging items.

Usage

Use the **status** command to list all breakpoints, actions, displays, and tracings. This listing displays the debugging item number needed for the **delete** command.

Zombied items

If the **delete** command has been invoked on a debugging item that is currently active on the execution stack, **status** reports the item as **zombied**. When execution continues, the zombied item will be deleted once it has completed executing, and **status** will no longer list it.

See Also

action, **delete**, **display**, **stop**, **trace**, **when**

step

steps execution by statement, entering functions

cdm	pdm
✔	✔

Command syntax

step

step *number*

Description

<< *none* >>	Executes a single statement and then stops execution. Motif and OPEN LOOK: Updates the Source area to display the new line of execution.
number	Executes the specified number of statements and then stops execution.

Options

The following options affect the **step** command:

cxx_suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.
src_step	(Ascii ObjectCenter only) Specifies the number of lines of source code to be displayed after execution of a statement.

See the **options** entry for more details about each option. ObjectCenter does not support thsese options in process debugging mode (**pdm**).

Usage

Use the **step** command to single-step through your program, going into functions when they are called. If a line contains multiple statements, execution moves to the next statement on the line.

Automatic mode switching

As you step through code, ObjectCenter automatically matches the Workspace mode to the language type of the module you are currently in. For example, if the current mode is C++ and you use **step** to move into a C module, then ObjectCenter automatically changes to C mode.

Threaded applications

In threaded applications, **step** executes one statement of the specified thread. If **step** skips over a function (in the case of a function without debug information), LWPs continue to the end of skipped function calls.

NOTE Debugging of threaded applications is currently only supported in process debugging mode, and it is not supported on all platforms. Please refer to the "Product limitations" section in the "About This Release" appendix to the online *ObjectCenter Reference* for more information.

Example

Here are two examples of using the **step** command in a threaded application. In the first example, the active thread is in a function without debugging information.

```
pdm (break 1) 13 -> thread -info
>   t@5   a l@1 thread_bounce__FP13DrawableShape()running in
_thrsys_poll()
pdm (break 1) 14 -> step
Single stepping until exit from function _thrsys_poll, which has no line
number information.
Stopped in function: `_poll'. No source file info.
pdm (break 1) 15 -> thread -info
>   t@5   a l@1 thread_bounce__FP13DrawableShape()running in _poll()
```

In the second example, we use the thread command to make **t@6** the current active thread and then step through it. We use the **thread -info** command to see where the thread is after each step, and the where command to see the stack trace.

```
pdm (break 1) 36 -> thread t@6
pdm (break 1) 36 -> thread -info
>   t@6   a l@4 thread_bounce__FP13DrawableShape()running in
bounce__13DrawableShapeFv()
pdm (break 1) 37 -> step
pdm (break 1) 38 -> thread -info
>   t@6   a l@4 thread_bounce__FP13DrawableShape()running in
doDraw__13DrawableShapeFv()
pdm (break 1) 39 -> step
pdm (break 1) 40 -> thread -info
>   t@6   a l@4 thread_bounce__FP13DrawableShape()running in
createTable__13DrawableShapeFv()
pdm (break 1) 41 -> step
pdm (break 1) 42 -> where
#0  0x12440 in DrawableShape::createTable(void) (this=0x27c48) at
shapes.C:164
#1  0x122c0 in DrawableShape::doDraw(void) (this=0x27c48) at shapes.C:110
#2  0x123d0 in DrawableShape::bounce(void) (this=0x27c48) at shapes.C:132
#3  0x11b50 in thread_bounce(DrawableShape *) (shape=0x27c48) at
mainTh.C:12
```

Restrictions

The **step** command does not stop inside object code functions that do not have debugging information (functions either compiled without the **-g** switch or loaded with the **-G** switch).

The **step** command does not stop in functions that initialize static variables.

See Also

next, **stepout**, **stepi**

stepi

steps execution in machine instructions by statement, entering functions

cdm	pdm
	✔

Command syntax

stepi

stepi *number*

Description

<< *none* >>	Executes a single machine instruction and then stops execution.
number	Executes the specified number of machine instructions and then stops execution.

Usage

Use the **stepi** command to single-step through the machine instructions in your program, going into functions when they are called. If a line contains multiple statements, execution moves to the next statement on the line.

See Also

listi, **nexti**, **step**, **stopi**

stepout

continues execution until the current function returns

cdm	pdm
✔	✔

Command syntax

stepout

Description

<< none >> Continues execution until the current function returns and then stops execution at the next statement in the calling function.

Options

The following ObjectCenter option affects the **stepout** command:

src_step (Ascii ObjectCenter only) Specifies the number of lines of source code to be displayed after execution of a statement.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **stepout** command to move execution to the point where the current function returns. This command is particularly useful if you inadvertently step into a function and want to continue stepping through the calling function.

Restrictions

The **stepout** command does not work when you are stopped in object code in component debugging mode.

See Also

next, **step**

stop

sets a breakpoint

cdm	pdm
✔	✔

Command syntax

stop
stop if *cond*
stop [at] *line*
stop at *line* **if** *cond*
stop [in] *func*
stop [in] *func* **if** *cond*
stop [at] *"file":line*
stop [on] *address*
stop [on] *lvalue*
stop [on] *variable*

Description

<< *none* >>	In process debugging mode, sets a breakpoint at the current location. Displays a stop sign next to the line containing the breakpoint in the Source area. In component debugging mode, creates a break level and stops execution immediately; no breakpoint is set. Useful in the form of its equivalent ObjectCenter function call, **centerline_stop(" ")**. Motif and OPEN LOOK: Displays a stop sign next to any line containing a breakpoint, if the file is listed in the Source area.
if *cond*	Creates a break level and stops execution if *cond* is true, where *cond* is a Boolean expression. (pdm only)

[**at**] *line*	Sets a breakpoint at the specified line in the current file. In pdm**,** specifying **at** is required, rather than optional.
at *line* **if** *cond*	Sets a breakpoint at the specified line in the current file if *cond* is true, where *cond* is a Boolean expression. (pdm only)
[**in**] *func*	Sets a breakpoint at the first line of the specified function. In pdm**,** specifying **in** is required.
[**in**] *func* **if** *cond*	Sets a breakpoint at the first line of the specified function if *cond* is true, where *cond* is a Boolean expression. (pdm only)
[**at**] *"file":line*	Sets a breakpoint at the specified line in the specified file. (cdm only)
[**on**] *address*	Sets a breakpoint at the specified address. Stops execution whenever the byte at the specified address is modified.The *address* argument must be a hexadecimal value. (cdm only)
[**on**] *lvalue*	Sets a breakpoint on the referenced address. Stops execution whenever the referenced address is modified. The *lvalue* argument is any C++ or C lvalue, such as a dereferenced pointer. (cdm only)
[**on**] *variable*	Sets a breakpoint on a variable. Stops execution whenever the variable is modified.(cdm only)

Options

The following ObjectCenter options affect the **stop** command:

src_stop	(Ascii ObjectCenter only) Specifies number of lines of source code to be displayed when a break level is first created.
save_memory	Using this option means you cannot use **stop on** *variable.*

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **stop** command to set a breakpoint in your program's code. When the breakpoint is encountered, execution is interrupted and a break level is created.

In addition to setting breakpoints in source code, you can set breakpoints in code that is loaded in object form. If the object code contains debugging information from the compiler (that is, if the object code was compiled using the **-g** switch *and* was loaded into ObjectCenter *without* the **-G** switch), then you can set a breakpoint at a line, at a line in a specified file, or in a function. Breakpoints *cannot* be set on an address, lvalue, or variable in object code.

In object code loaded without debugging information (either compiled *without* the **-g** option or loaded with the **-G** option), you can set a breakpoint on a function name, but you cannot set a breakpoint on a particular line of code.

To continue execution after the breakpoint, use the **cont** command.

You can also set a conditional breakpoint with the **action** command in component debugging mode or the **when** command in process debugging mode. To remove a breakpoint, use the **delete** command. To view a list of all breakpoints, use the **status** command.

NOTE When you save your project to a project file, breakpoints may not be saved in the form in which you entered them. For example, if you set a breakpoint in a function, the breakpoint is set on the file and line number at which the function occurs rather than on the function name. As a result, breakpoints may not behave in the way you expect them to when you reload your project.

Restrictions

Breakpoints set on addresses that are modified while executing in object code are not performed.

NOTE See the *ObjectCenter Platform Guide* for information about setting breakpoints in shared libraries in pdm.

See Also

action, **cont**, **delete**, **status**, **stopi**, **when**

stopi

sets a breakpoint at a machine instruction

cdm	pdm
	✔

Command syntax

stopi

stopi [at] *address*

Description

<< *none* >>	Sets a breakpoint on the current location's address.
[at] *address*	Sets a breakpoint on the specified address. Stops execution whenever the byte at the specified address is modified. The *address* argument must be specified as a numeric string.

suppress

suppresses reporting of a warning

cdm	pdm
✔	

Command syntax

suppress

suppress *num*

suppress *num* **[at]** *line*

suppress *num* **[at]** *"file":line*

suppress *num* **[in]** *directory*

suppress *num* **[in]** *file*

suppress *num* **in** *function*

suppress *num* **[in]** *lib***(***module***)**

suppress *num* **[on]** *identifier*

suppress *num* **[on]** *function*

suppress save [*file***]**

Description

<< *none* >>	Lists all currently suppressed violations.
num	Suppresses reporting of the specified violation everywhere.
num **[at]** *line*	Suppresses reporting of the specified violation at the specified line.
num **[at]** *"file":line*	Suppresses reporting of the specified violation at the specified line in the specified file.
num **[in]** *directory*	Suppresses reporting of the specified violation in all files in the specified directory or in any subdirectories of the specified directory.

num **[in]** *file*	Suppresses reporting of the specified violation in the specified file.
num **in** *function*	Suppresses reporting of the specified violation while in the specified function. The function name must include the signature for C++ functions, for example `-> suppress 42 in testfn(void)`
num **[in]** *lib***(***module***)**	Suppresses reporting of the specified violation in the specified module of the specified library. For example, the following suppresses reporting of message 731 in the library module **sel_common.o**: `-> suppress 731 in /usr/lib/libsuntool.a(sel_common.o)`
num **[on]** *identifier*	The *identifier* argument is any variable, typedef, class / struct / union tag, or macro name. Suppresses reporting of the specified violation if the violation involves the specified identifier.
num **[on]** *function*	Suppresses reporting of the specified violation when the specified function is called.
save [*file***]**	If a file is specified, writes a list of all currently suppressed violations in the file. If a file is not specified, prompts for a filename before saving.

Options

The following ObjectCenter option affects the **suppress** command:

terse_suppress	Tells the **suppress** command not to echo the name of the violation being suppressed.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **suppress** command to suppress the reporting of ObjectCenter violations (warnings and errors). Use the Manual Browser to view the "violations" topic for a list of the violations that ObjectCenter reports; you can invoke the Manual Browser by issuing the **man** command in the Workspace.

Saving and reusing a set of suppressions

By using **suppress** with the **save** *file* argument, you can save the suppressed violations to a file and then in another session use the **source** command to read in the suppressions from the file. Suppressions that you use **source** to read in from a file are added to any suppressions that are current. Another way to retain a set of suppressions for later use is to save a project file.

Handling lint comments

If the comment /*SUPPRESS *num [args]* */ appears in source code, static error checking is suppressed for the specified violation. If the comment appears at the global level of a file, the violation is suppressed for the entire file. If the comment appears within a function, the violation is suppressed only for the following line.

See Also

source, **unsuppress**

suspend

suspends ObjectCenter and returns to the shell

cdm	pdm
✔	

Command syntax **suspend**

Description

<< *none* >>	Suspends ObjectCenter and returns to the shell. (Ascii ObjectCenter only)

Usage

Use the **suspend** command to suspend ObjectCenter and return to the shell. The **suspend** command is useful for creating scripts that will suspend ObjectCenter or for situations when entering Control-z will not produce a stop signal.

Use **fg** to return to ObjectCenter after being suspended.

See Also **quit**, **save**

swap

replaces a file or function with its source / object counterpart

cdm	pdm
✔	

Command syntax

swap *file*

swap *function*

Description

file — If the specified file was loaded as source code, unloads the specified source file and loads the corresponding object file.

If the specified file was loaded as object code, unloads the specified object file and loadsthe corresponding source file.

function — Unloads the entire file containing the specified function and loads the corresponding source / object counterpart.

Switches

-C — Swaps in the specified file, or the file containing the specified function, as a C module.

-CXX — Swaps in the specified file, or the file containing the specified function, as a C++ module.

Options

The following ObjectCenter options affect the **swap** command:

c_suffixes — Specifies file extensions to search for when ObjectCenter needs to find a C source file that corresponds to a given object file.

cxx_suffixes — Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.

path	Specifies the search path for loading source and object files (not for **#include** files). You must also set the **swap_uses_path** option for the **path** option to affect the **swap** command.
swap_uses_path	Determines whether the **swap** command uses the **path** option when looking for files.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **swap** command to do either of the following:

- Replace a file or function loaded as source code with the corresponding object code.
- Replace a file or function loaded as object code with the corresponding source code.

The **swap** command with the *function* argument is particularly useful for replacing library functions with their source counterparts, when the library is not a shared library. When a library function is replaced, only the object module containing the function is replaced, not the entire library.

You cannot use **swap** to replace a shared library function with the corresponding source file. Instead, you must unload the entire shared library, load the source file, and load the shared library again for the remaining object files.

Like **swap**, the **load** command can also be used to exchange source and object code.

Restrictions

In library files, the names of constituent object files are truncated to 15 characters. When **swap** attempts to swap a library module with a truncated name, it displays “Unknown suffix”.

The **swap** command will not replace a source module with an object module in a library. To do so, **unload** the source module (or a function in it) and **load** the library, if it is not already loaded, and issue the **link** command.

See Also

load, **unload**

templates

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

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

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

In most cases, you have to follow a few simple rules to use templates in ObjectCenter; the automatic instantiation process takes care of all the complicated details. If you are already familiar with C++ templates, and you want a quick description of their implementation in ObjectCenter, see the "Summary of usage" section on page 382.

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

- Basic concepts and syntax
- Using templates with ObjectCenter
- The instantiation process
- Coding conventions
- Lookup schemes
- Map files
- Switches for templates
- Usage scenarios
- Specializations
- Examples
- Common pitfalls

- Troubleshooting
- Tools
- Summary of terminology

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

Basic concepts and syntax

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

Class templates

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

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

Creating the class from the template is called instantiating the template. See Figure 5 for a conceptual illustration of the relationship between a class template, an individual class (template class) instantiating the template, and declaring an object of the template class.

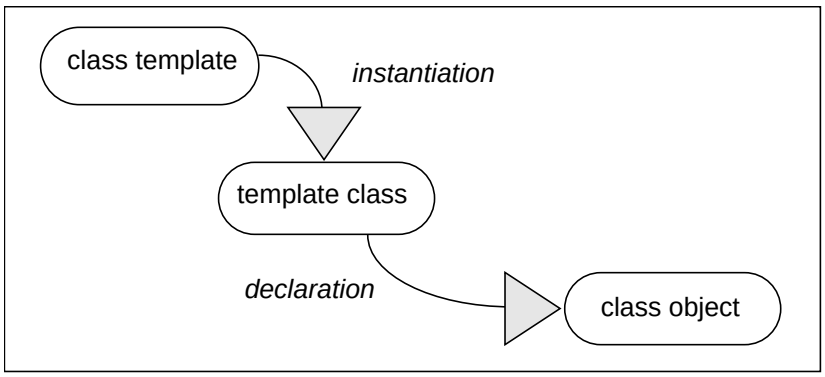

Figure 5 Instantiation and Declaration: From Template to Class to Object

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

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

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

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

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

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

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

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

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

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

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

See Figure 7 for a conceptual illustration of the instantiation and declaration of a class template in the bank example.

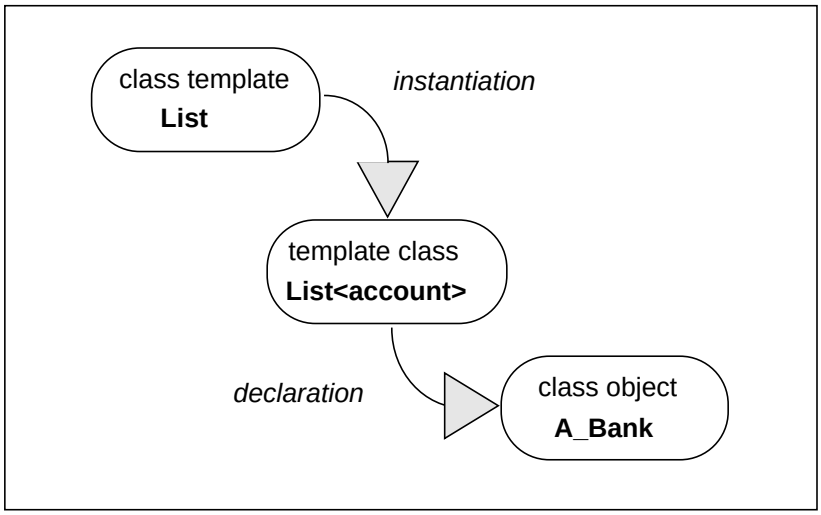

Figure 6 From **List** Class Template to **A_Bank** Class Object

Function templates

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

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

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

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

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

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

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

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

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

See Figure 7 for a conceptual illustration of the instantiation and use of the **max** function template with **Circle** as the parameterized type. Note that the **Circle(max)** function is generated implicitly by the compiler.

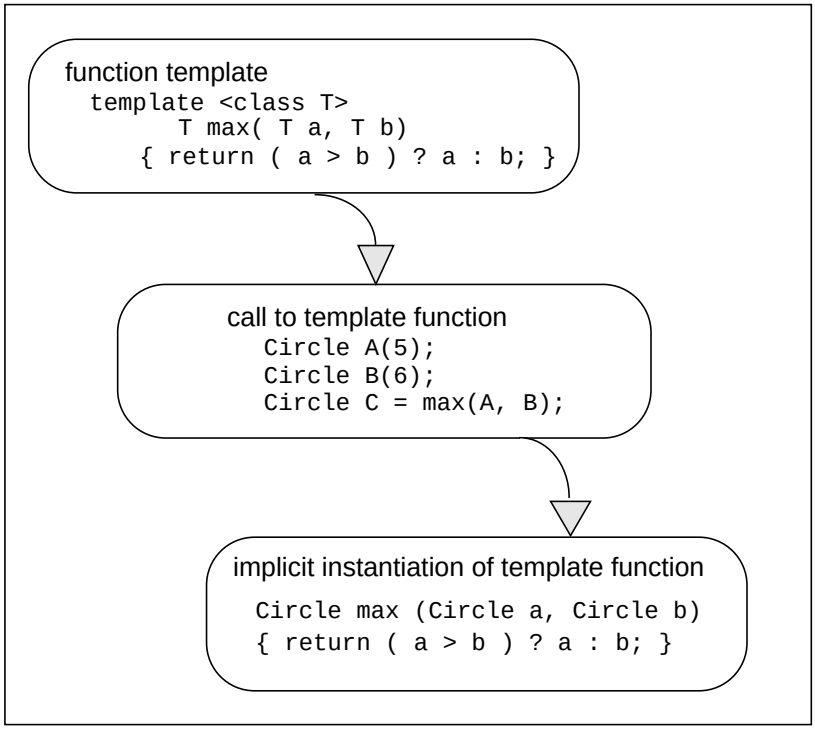

Figure 7 From Function Template to Function Call

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

In the case illustrated in Figure 7, the parameterized type in the function call is **Circle**, so the instantiated template function returns a **Circle** and takes **Circles** as arguments.

Using templates with ObjectCenter

In this section we present an overview of the use of templates with ObjectCenter; we do so by focussing on another example of a simple class template. Later in the chapter we describe the instantiation process and coding conventions in more detail. The three files used in this example, **Vector.h**, **Vector.c**, and **appVector.C**, are available online in the examples directory.

NOTE The example in this section is a simple example that assumes all files are in one directory; see the "Using options to control the template instantiation process" **TIP** on page 363 for more information about using options and switches to tell ObjectCenter how to find files in other directories.

Declaring a template in a .h file

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

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

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

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

NOTE The template instantiation system allows you to use filenames with different suffixes. See the “Dynamic extension lookup” section on page 355 and the “Map files” section on page 357 for details on how to do so correctly.

Specifying a template implementation in a .c file

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

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

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

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

Note that the code in **Vector**'s declaration and implementation is parameterized — it uses a type that is unknown but represented by **T**. Also, note that the **.c** file does not include the **.h** file. The automatic instantiation system will locate **Vector.c** according to the rules described in 'Dynamic extension lookup' on page 355.

Specifying a template class in an application

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

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

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

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

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

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

Loading an application in ObjectCenter

To load the **Vector** application into ObjectCenter, issue the following command in the Workspace:

```
-> load appVector.C
```

NOTE Do not load the definition file, **Vector.c**, or the declaration file, **Vector.h**.

ObjectCenter can report many kinds of syntax errors when you load your application. However, it cannot report errors in template definitions until it attempts to load the instantiated templates as source or as object code; we describe this step in the next section.

Linking and running your program

After you load your application, issue a **link** or **run** command in the Workspace. Each of these commands tells ObjectCenter to create object files for the template classes as needed by your application:

```
-> link
```

Issuing the **link** command causes ObjectCenter to invoke the **CC** command on the instantiation source file, causing the creation of an object instantiation file, which it then loads into the environment. Here's an example for the **Vector** case:

```
-> link
Linking from ...
Loading: ptrepository/Vector__pt__2_i.o
```

In the case of the **Vector** example, the object file instantiating **Vector<int>** contains the template class members **Vector<int>::Vector()** and **Vector<int>::operator[](int)**. This process is called **instantiation**.

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

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

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

```
-> run
Executing: a.out
Program exiting with return status = 0.
```

and in the Run Window:

```
1 1
2 4
3 9
4 16
5 25
```

Instantiation in the ObjectCenter environment

In most cases, you can use templates in ObjectCenter without knowing the details of instantiation. However, if you're interested, or if you want to change the defaults that ObjectCenter uses, read this section for more details.

ObjectCenter stores the files it needs for template instantiation in the template **repository**. The repository is a directory that can contain a **.o** file, which is the template instantiation object file, along with a **.c** (instantiation source file), a **.cs** (checksum file), a **.he** file (contains information about header file dependencies—see on page368) and the default name mapping file, **defmap**.

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

The name mapping file contains information about templates, including the names of the files where the templates are declared and instantiated. See the "Map files" section on page 357 for more information. You may find it helpful to look at the contents of each of the files created in the repository directory before you read the following discussion of the instantiation process.

By default, ObjectCenter loads instantiation modules as object files. If you want them loaded as source, unset the **tmpl_instantiate_obj** option as shown:

```
-> unsetopt tmpl_instantiate_obj
```

See Figure 8 for a conceptual overview of the steps in instantiation in the ObjectCenter environment.

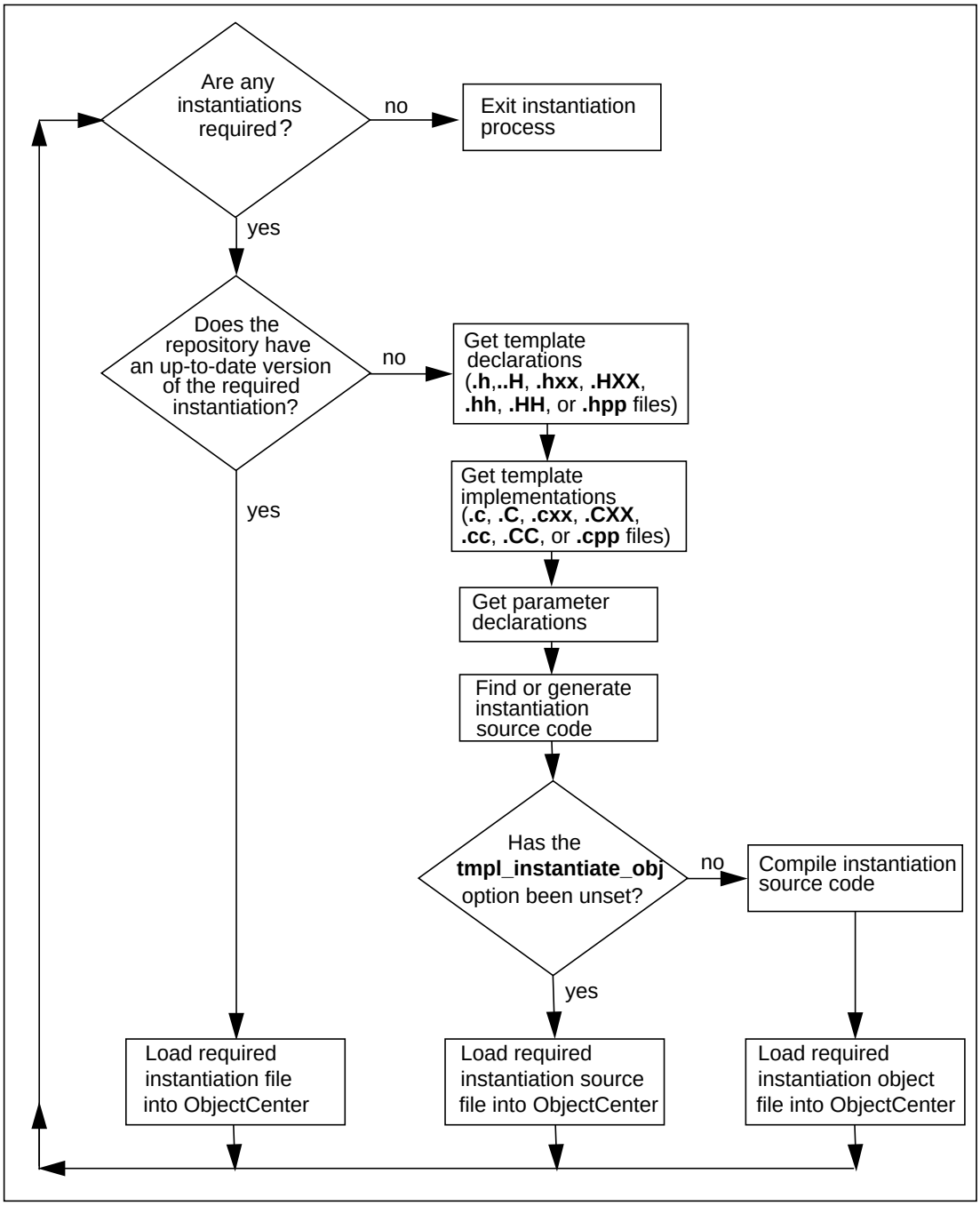

Figure 8 Steps in the Template Instantiation Process in the ObjectCenter Environment

Here's a more detailed explanation of how instantiation works:

- ObjectCenter checks the repository for an up-to-date version of the required source or object file. If the instantiation for the template class is already available, ObjectCenter does not recreate it.

  As part of this step, ObjectCenter checks the information in the **.he** file in the repository to see if any header files required by the instantiation file have become outdated.

- Alternatively, ObjectCenter searches the name mapping file (or files, if you have specified more than one) to find files for the template declaration as well as the declarations of any types used by the instantiation.

  To find the template definition file for a given template, ObjectCenter uses the algorithm described in 'Dynamic extension lookup' on page 355.

  You can override the default name of the definition file by specifying a different name in a user-defined name mapping file in the repository. See the "Specifying user-defined map files" section on page 359.

- As it finds template declaration and definition files and all required parameter declaration files, ObjectCenter uses them to find or generate instantiation source.

- By default, ObjectCenter compiles the instantiation source, creating the object file for the template class.Then ObjectCenter loads the instantiation object module into the ObjectCenter environment.

- You can change the default and load source instead of object by unsetting the **tmpl_instantiate_obj** option; if you do so, ObjectCenter loads the instantiation source file into the ObjectCenter environment.

Instantiation outside the ObjectCenter environment

There are some differences between the way template instantiation works inside the ObjectCenter environment and outside the environment. See Figure 8 for a conceptual illustration of instantiation outside the environment. You might want to compare this figure with Figure 8 on page 346, which illustrates instantiation inside the environment.

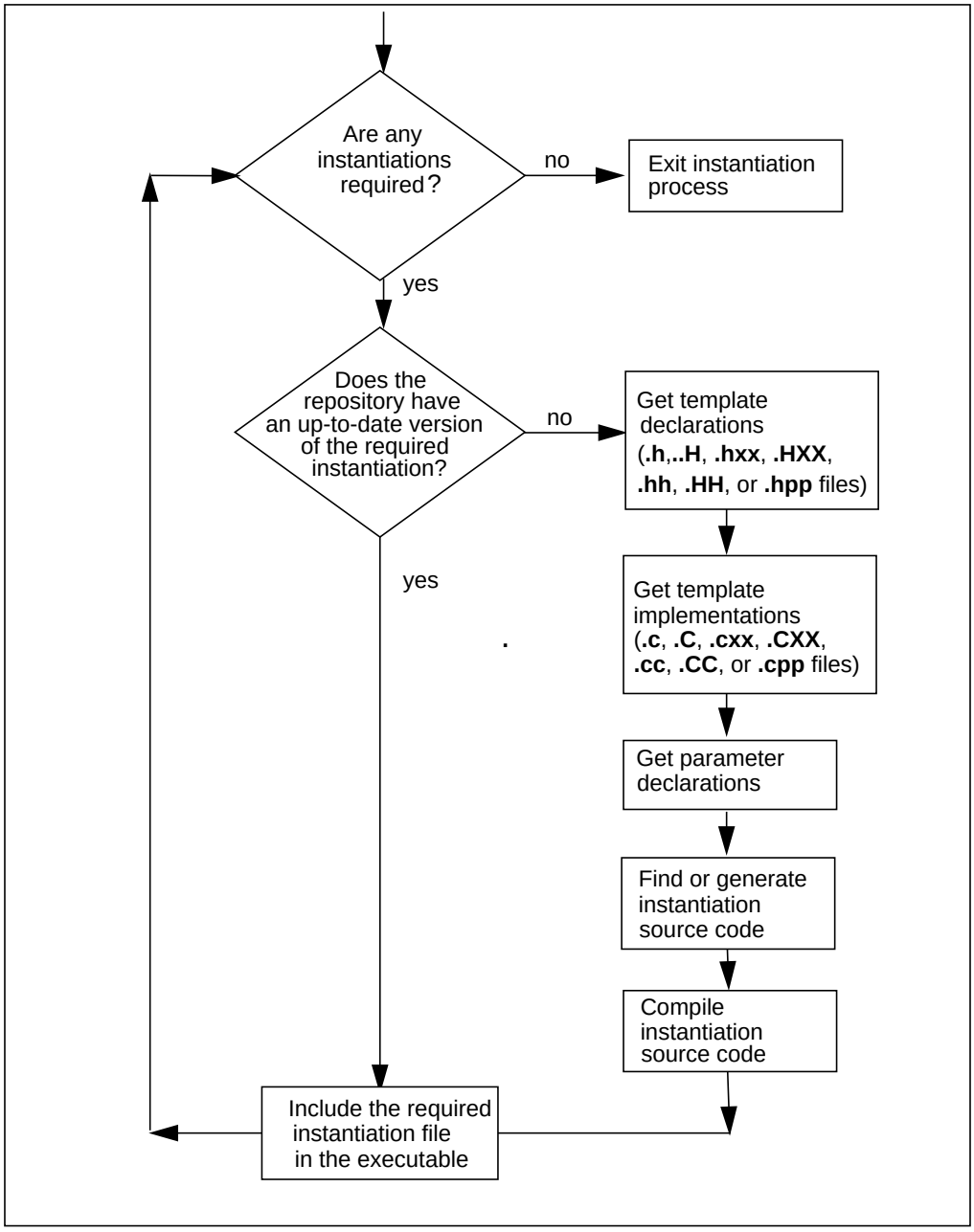

Figure 9 Steps in the Template Instantiation Process with **CC**

The following steps summarize the instantiation process. For a more detailed explanation, see "Template Instantiation in C++ Release 3.0.2", an excerpt from the *AT&T Language System Selected Readings*, which is available online in the **CenterLine/***archos***/docs** directory.

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

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

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

   The template definition file (that is, the file containing the implementation of the template) is assumed to be a file with the same name as the template declaration file, except that it has a different suffix. The mechanism for looking up the template types is described on on page356.

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

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

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

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

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

Major differences

See Table 29 for an overview and description of the major differences.

Table 29 Differences between Template Instantiation in ObjectCenter vs. **CC** (outside of ObjectCenter)

Areas of Difference	In the ObjectCenter Environment	Outside the ObjectCenter Environment
When does instantiation occur?	Auto-instantiation incrementally occurs whenever your code contains templates and you: • Issue **link** command • Issue **run** command • Invoke a function, such as a constructor, that references the template functions or template member functions.	Auto-instantiation occurs at link time, if your code uses templates. If you are compiling and linking in one step, **CC** does not use auto-instantiation.
What switches affect instantiation?	See the "Using options to control the template instantiation process" **TIP** on page 363. Also, the **-ptn** switch has no effect in ObjectCenter.	Instantiation modules are compiled using switches that are specified on the **CC** command line.
When is the name mapping file updated?	Loading a source file that is a template instantiation module does not affect the name mapping file.	Compiling a file causes the name mapping file in the repository to be updated.
When can you load specializations?	Specializations must be loaded before auto-instantiation occurs.	Specializations can be provided at link time.

Generally speaking, the **load** operation in ObjectCenter corresponds to compiling outside the environment, and the **link** operation inside the environment corresponds to linking object files outside the environment into an executable. One of the chief differences between template instantiation inside and outside of ObjectCenter results from the fact that ObjectCenter never combines the loading and linking process, whereas the **CC** command outside the environment does combine them.

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

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

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

Some switches related to compiling are as follows:

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

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

-g +d

Using CC with templates in header files

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

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

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

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

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

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

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

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

Using automatic tools and make

As we mentioned previously, when you load an application that uses templates, you do not explicitly load the template definition file; ObjectCenter's instantiation process automatically loads the correct file just because your application uses the template. Also, ObjectCenter automatically recompiles the template definition file if necessary because it has become outdated.

However, outside the environment, you need to specify the file that contains the template implementation (the template definition file) as a dependency in your makefile. And, if you use automatic tools that check for dependencies, you have to manually indicate that template definition files are dependencies. For example, if app.C depends on the template.h definition file, your makefile should contain these lines:

```
app.C:
#include <template.h>

app.o: template.h template.c
```

Coding conventions

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

Argument declaration files

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

An argument type should be declared in a single header file that is self-contained or that includes other headers that it needs. If this is not possible then you must write a map file; see the "Map files" section on page 357.

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

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

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

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

Include guards

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

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

Template declaration files

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

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

For function templates, use a forward declaration:

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

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

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

Template definition files

A template definition file contains the implementation of a template: the definitions of member function templates and initializers for static members of the template. The definition file has the same name as the template declaration file, but with a different suffix. See 'Dynamic extension lookup' on page 355 for a list of suffixes.

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

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

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

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

In general, a definition file should not include the declaration file that matches it, nor the argument declaration files that declare any template argument types, unless you use include guards consistently as recommended on on page353. Including a guarded template definition file in a template declaration file will cause the definition file to be typechecked at application compile time, at the expense of slower compiling.

There must be a definition file for every declaration file, or a map file that overrides the standard convention. User-defined map files are described in 'Specifying user-defined map files' on page 359. If a template definition file does not exist along the **-I** path, ObjectCenter issues a warning and does not include the file. All other missing files will cause a preprocessor error at instantiation time.

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

Inline functions

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

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

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

Type lookup

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

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

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

Dynamic extension lookup

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

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

Finding template parameters

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

Finding template types

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

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

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

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

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

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

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

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

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

or the declaration file names, it issues a warning and does not include a header file in the instantiation file.

Map files

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

The default map file

The default name for the current name mapping file is **defmap**; the preceding version is named **defmap.old**. This map file is maintained by ObjectCenter, and you should not edit it. You can override the **defmap** file by specifying a user-defined name mapping file, as described on on page359.

Map files for the Vector example

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

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

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

The entry beginning with **@dec** shows that type **Vector** is *declared* in **Vector.h**. If you look at the whole file, you will see that there is no **@def** entry specifying where **Vector** is *defined* (implemented). The instantiation system uses the algorithm on on page356 to locate the implementation for **Vector** in **Vector.c**.

In these lines:

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

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

Sharing a repository

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

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

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

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

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

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

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

Encoding of functions in map files

In map files, operator function templates are encoded as described in Section 7.2.1c in *The Annotated C++ Reference Manual* . For example, operator << is encoded as **__ls**. Also, function template types are recorded without parameter information; as a result they appear as a single map file entry.

Specifying user-defined map files

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

You can create user-specified map files to override the lookup mechanism described in 'Type lookup' on page 355.

Example of overriding default filenames

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

nmap1:

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

Specifying application files in map files

Note that a map file entry does not have to specify application files; an entry without any applications serves as a last resort if the type cannot otherwise be found. Typically, application files are recorded to handle the case where there are distinct applications sharing one repository. Also, you don't have to use **@0**, **@1** and so on as shown in 'Sharing a repository' on page 358; you can spell the application name out. For example, instead of this:

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

you can write this:

```
@dec account banking account
```

Source and object file basenames

The cfront instantiation mechanism expects the basename of the source file that references a template to have the same name as the object file for the module. The basename that appears in the **defmap** file is that of the source module, but **ptlink** needs the name of the object module.

Suppose your source file is called **main.C**, but the corresponding object module is **test.o**. The beginning of the defmap file looks like this:

```
@tab
main
@etab
```

ptlink searches for an object module called **main.o** and fails to find the information it needs in **test.o**. It then assumes that the template declaration and definitions are to be found in files with the same basename as the template type name. If this isn't the case, **ptlink** issues a fatal error.

If you are unable to use the same basename for source and object modules, create a user-defined map file containing the object file's name. Copy the **defmap** file generated by the failed instantiation to **nmap001**, then replace the source module name in the **@tab** section with the object module name and recompile.

A similar problem can occur with modules linked in from a library archive on platforms on which the **ar** command truncates filenames. If **ptlink** fails in this situation, either shorten the basename of the file in the library, or include the name of the object module in the link line.

Switches for templates

Table 30 describes the switches used by the ObjectCenter template instantiation system.

Table 30 Template Instantiation Switches

Name of Switch	What the Switch Does
-pta	Directs ObjectCenter to instantiate the whole template, rather than only those members that are needed.
-ptd*pathname*	Dumps list of instantiation objects to a file if any were recompiled or if the file does not exist. Also bypasses actual link step. Can be used with **-pti** in makefiles of this form:

```
appl:  appl.o ilist
       CC -pti -o appl ‘cat ilist‘ appl.o

appl.o: appl.c Vector.h A.h C.h

        CC -c appl.c

ilist:  always

        CC -ptdilist appl.o

always:
```

Table 30 Template Instantiation Switches (Continued)

Name of Switch	What the Switch Does
-ptf	Forces ObjectCenter to instantiate templates when the source file is loaded, instead of later, even if the program consists of more than one file. Forces **CC** to instantiate templates when the source file is compiled. We do not recommend that you use this switch with applications that comprise more than one file.
-pth	Forces repository names to be less than 14 characters even if the operating system supports long names. This is useful in building archive libraries.
-pti	Ignores **ptlink** pass.
-ptk	Forces **ptlink** to continue trying to instantiate even after instantiation errors on previous template classes.
-ptm*pathname*	Have **ptlink** dump out a “link map” showing what actions the link simulator took.
-ptn	Changes the behavior of one-file programs to work like multi-file programs. If you do not set this switch for a one-file program, then by default, all templates are instantiated. See 'Simple programs' on page 365 for more information.
-pto*pathname*	Consider instantiation modules in *pathname* to be out of date, and regenerate and compile. No checking is performed.
-ptr*pathname*	Specifies *pathname* as a repository. By default, *pathname* is **./ptrepository**. You can specify more than one repository by using the switch more than once; use the switch for each repository. If multiple repositories are specified, only the first is writable; the others are used to retrieve instantiation modules rather than store them as written. For example, **-ptr** might refer to a central project directory or a class library repository.

Table 30 Template Instantiation Switches (Continued)

Name of Switch	What the Switch Does
-pts	Splits instantiations into separate object files, with one function per object (including overloaded functions), and all class static data and virtual functions grouped into a single object.
-ptt	Use timestamps to determine when instantiations must be compiled. This switch is on by default.
-ptv	Specifies verbose mode for template instantiation; ObjectCenter announces each step in the instantiation process. This is especially useful when you're learning about templates.

Options

Use the **tmpl_instantiate_obj** option to specify that you want template instantiation modules to be loaded into ObjectCenter as source rather than object. By default, these modules are loaded as object. Use the following command line to change the default:

```
-> unsetopt tmpl_instantiate_obj
```

Use the **tmpl_instantiate_flg** option to specify the switches that you want ObjectCenter to use when loading template instantiation modules. For instance, use the following command to specify that you always want verbose mode for template instantiation:

```
-> setopt tmpl_instantiate_flg -ptv
```

TIP: Using options to control the template instantiation process

ObjectCenter provides several options that influence the compilation and / or loading and / or linking of source code. In case you are confused about the use of these options with templates, here are some guidelines:

- Use the **cxxargs** option to specify arguments to be passed to the **CC** command, whenever a source module is compiled; this applies to template instantiation source modules as well as other C++ source files.
- The **sys_load_cxxflags** option specifies switches indicating the search paths for system libraries and **#include** files. These switches are always passed to the **load** command when loading template instantiation modules as well as other C++ files.
- Use the **load_flags** option to specify search paths for libraries and **#include** files that are needed for your application that are not specified in **sys_load_cxxflags**. Make sure that the path to find each template declaration or definition file needed by your application is specified in either the **sys_load_cxxflags** option or **load_flags**. Also, use the **load_flags** option to specify other switches related to template instantiation that are needed at load time; these switches are **-ptf, -ptr**, and **-ptv.** See Table 30, “Template Instantiation Switches,” on page 360 for more details about these switches.
- Use the **tmpl_instantiate_flg** option to specify switches related to template instantiation that are related to linking the instantiation modules. These switches are as follows: **-pta**, **-ptr**, **-pts**, and **-ptv**.
- Use the **tmpl_instantiate_obj** option to tell ObjectCenter whether template instantiation modules should be loaded as source or object. By default, this option is set, which means load template instantiation modules as object. The **tmpl_instantiate_obj** option has no effect on the switches to be used when a template instantiation source module is compiled or loaded. If the **tmpl_instantiate_obj** option is set, ObjectCenter uses the switches specified in **cxxargs** to compile the instantiation modules.

Here are several examples of fragments from possible makefiles:

```
Example 1:
   #load $(CXXFLAGS) app.C ...
   #setopt tmpl_instantiate_flg -ptv -ptr my_repos $(CXXFLAGS)
   #link

Example 2: (tmpl_instantiate_obj is TRUE, templates compiled using
cxxargs and then loaded)
   #setopt load_flags $(CXXFLAGS)
   #load app.C ...
   #setopt cxxargs $(CXXFLAGS)
   #link

Example 3: (tmpl_instantiate_obj is FALSE, templates loaded as source
using load_flags)
   #setopt load_flags $(CXXFLAGS)
   #load app.C
```

Using templates in the Workspace

Just as with other C++ language features, ObjectCenter supports the declaration, definition, and use of templates in the Workspace.

Specifying a template class object in the Workspace

For example, if you have loaded the Vector example as previously described, you could enter the following in the Workspace:

```
-> Vector<int> my_vector;
(class Vector<int> *) 0x32dfc0 /* (class
Vector<int>) my_vector */
```

Similarly, you can use the Vector template with other parameterized types, such as a **float**:

```
-> Vector<float> my_other_vector;
CC +g +d Vector__pt__2_f.c:
cc -c -g -Dsun -DBSD -Dsparc -Dunix
-I/usr/include/X11R4 -I/s/apps/openwin3.0/include
-I/tmp_mnt/hosts/plough/u5/incl -I/usr/include
-L/usr/lib/X11R4 -L/s/apps/openwin3.0/lib
-L/tmp_mnt/hosts/plough/u5/lib -L/lib -L/usr/lib
-L/usr/local/lib Vector__pt__2_f.c
Loading: ptcrepository/Vector__pt__2_f.o
(class Vector<float> *) 0x32e000 /* (class
Vector<float>) my_other_vector */
->
```

Defining templates in the Workspace

You can also use the Workspace to try out implementations for templates that you have declared but not yet defined.

For instance, suppose you have declared a template class in **my_template.h**, and you have developed an application using the template class in **my_template_app.C**. Your next development task is to develop the definition of **my_template**, which will eventually be listed in **my_template.c**.

You can try out definitions for **my_template** in the Workspace by following these steps:

1. Enter the following in the Workspace:

   ```
   -> load_header "my_template.h"
   -> load my_template_app.C
   ```

2. Type definitions for the members of **my_template** in the Workspace.

3. Run the program.

If you define templates in the Workspace, ObjectCenter instantiates them without using the automatic instantiation process described in the "Instantiation in the ObjectCenter environment" section on page 345.

Usage scenarios

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

Simple programs

By a simple program we mean a one-file program that contains all the templates and argument types it requires; such a program does not require a linking step when compiled outside the environment. However, by default, ObjectCenter invokes its template instantiation system for such simple programs. You can bypass this link-time instantiation system by specifying the **-ptf** switch, which forces all templates to be instantiated when the source module is loaded.

Small and medium projects

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

```
-> setopt tmpl_instantiate_flg -I/usr/template/incl
-> load -I/usr/template/incl file1.c
.
.
.
-> link
```

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

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

```
-> sh mkdir rep1
-> load -ptrrep1 file1.c
```

Repository permissions

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

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

```
% chmod 775 ptrepository
```

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

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

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

```
% sh umask 002
```

Large projects and multiple repositories

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

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

```
-> load -I/usr/jones/tincl -I/usr/proj/tincl -I/usr/jones/incl\
-I/usr/proj/incl -ptr/usr/jones/rep -ptr/usr/proj/rep file.c
```

Given the preceding command, when it instantiates templates used in **file.c**, ObjectCenter uses the following repositories:

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

Repository management

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

Sharing code and using archives

Instantiations in a repository are can be simply object files; it depends on whether the **tmpl_instantiate_obj** option is set. If they are object files,you can easily export them into an archive. For example, with the default repository one can say:

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

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

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

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

Libraries

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

For example, suppose you wanted to form closure for a library named **/usr/proj/lib.a**, and the ptrepository and other objects on which the library depends are in the directory **/my/lib_dir**. The following example extracts the old modules from the archive, recompiles with the **-pts** switch to **CC** to split the instantiations into separate files, removes the old library, and creates a new library archive including the object files from the template repository.

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

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

Note that the object filenames in the repository may be longer than the 14 characters that **ar** will handle. You can use the **-pth** option to limit names to 14 characters when you compile, or rename object files; a tool for this purpose is described in 'Tools' on page 380.

Using a library of template instantiations

If you want to use a library of template instantiations and forego the instantiation process, you must explicitly load the library before performing the link. Outside the ObjectCenter environment, you must link with the specified library, using the appropriate **-L** and **-I** flags explicitly,

Link-simulation algorithm

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

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

Note that object filenames in the repository may be longer than the 14 characters that **ar** will handle. You can use the **-pth** option to limit names to 14 characters when you compile, or rename object files; a tool for this purpose is described in the "Tools" section on page 380.

Dependency checking

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

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

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

---

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

Forcing reinstantiation

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

```
% touch template_name.suffix
```

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

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

Specializations

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

To use a specialization instead of the templates already mapped in the repository, you must load the source file containing the specialization before you start the automatic instantiation process. Note that the **-pta** and **-ptf** flags are incompatible with template specializations.

A specialization example

For instance, suppose you want to use the **appVector.C** application described in 'Using templates with ObjectCenter' on page 341. However, you want to override the implementation code in **Vector.c** for the case of integers as the parameterized type.

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

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

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

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

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

If you want to provide a specialization after an instantiation has been automatically provided, you must unload the instantiation, load the specialization, and link again.

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

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

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

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

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

Here's what happens in the Workspace when we load, link, and run this application in ObjectCenter:

```
-> load spec_vec.c
Loading (C++): spec_vec.c
-> load appVector.C
Loading (C++): appVector.C
-> link
Linking from '/usr/local/lib/libC.a' ..... Linking
completed.
Linking from '/usr/lib/libc.sa.1.6' ... Linking
completed.
CC +g +d Vector__pt__2_i.c:
cc -c -g -DBSD -Dunix
-I/tmp_mnt/hosts/plough/u5/incl -I/usr/include
-L/tmp_mnt/hosts/plough/u5/lib -L/lib -L/usr/lib
-L/usr/local/lib Vector__pt__2_i.c
Loading: ptrepository/Vector__pt__2_i.o
-> run
Executing: a.out
Linking from '/usr/local/lib/libC.a' ... Linking
completed.
Program exiting with return status = 0.

Resetting to top level.
```

And, in the window where we started ObjectCenter, the program's output is as follows:

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

Static template class data members

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

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

To specialize this, you could say:

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

somewhere in the application.

Examples

This section describes a few more small sample cases.

Single file

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

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

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

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

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

To execute this code in ObjectCenter, do the following:

```
-> load -ptf user_app.C
-> link
```

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

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

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

Separate compilation

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

Stack.h:

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

Stack.c:

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

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

userapp.C:

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

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

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

---

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

Here are the implementation details for the last example:

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

2. ObjectCenter determines that the following functions must be instantiated:

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

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

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

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

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

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

5. ObjectCenter compiles the instantiation source file, if necessary, and stores the resulting object file in the repository.

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

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

Specialization at link time

It is legal for a special case of a template member to be discovered at link time. For example, given the files shown in 'Separate compilation' on page 373, suppose this additional file were loaded into the environment before you issue the link or run command:

stringpop.c:

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

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

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

TIP: Avoiding the most common pitfalls when using templates

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

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

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

Keep in mind that you might get syntax errors during the final linking phase, since templates are instantiated later. If you do get syntax errors at the instantiation phase, edit only the template definition file, not your application file.

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

Troubleshooting

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

Network timestamps

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

External name length limitations

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

Map file problems

If you have many programs in the same directory using the same type name, for example, test cases using the type T, the default map file will become very large. You can compress the file by using a string table, as described in 'Sharing a repository' on page 358.

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

Violation of the one definition rule

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

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

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

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

Picking up the wrong versions of headers

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

Replaying source files

If a source file looks like this:

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

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

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

Function templates

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

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

has been seen.

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

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

```
$ CC -ptn prog.c
```

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

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

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

and this declaration of a specialization:

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

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

Static data member initialization

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

For example:

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

by itself would result in an unresolved external.

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

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

or this:

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

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

Type checking of template members

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

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

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

Renaming object files

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

The simplest solution to this problem is to write a map file with a type entry with no list of basenames (see 'Specifying application files in map files' on page 359).

Debug formats and large binaries

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

Tools

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

CenterLine/oc_2.0.0/*arch_os***/pt/tool1**

CenterLine/oc_2.0.0/*arch_os***/pt/tool2**

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

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

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

```
#!/bin/sh

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

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

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

exit 0
```

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

```
#!/bin/sh

# export contents of repository into an archive

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

pn=‘basename $0‘

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

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

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



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

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

rm -rf $t

exit 0
```

Summary of usage

Here's a brief summary of the steps required to load an application that uses templates:

1. Make sure that your template declarations and definitions are in files that are named according to the conventions in 'Dynamic extension lookup' on page 355, or that you override the conventions by following the instructions in the "Map files" section on page 357.
2. Use the **load** command to load your application, making sure that each template declaration or definition file is in a path specified with the **-I** switch in **load_flags** or in **sys_load_cxxflags**.
3. See the "Using options to control the template instantiation process" **TIP** on page 363 and "Using templates with ObjectCenter" section on page 341 for other switches related to templates.
4. Use the **tmpl_instantiate_flg** option to specify switches related to linking your template instantiation modules.
5. Issue the **link** command.
6. If ObjectCenter reports errors, fix the code in the appropriate template definition file.
7. Issue the **run** command.

See the "Avoiding the most common pitfalls when using templates" **TIP** on page 376 for more assistance.

Summary of terminology

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

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

defmap
: The default name for the name mapping file.

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

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

repository

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

specialization

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

template declaration

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

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

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

template definition

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

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

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

template definition file

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

template instance

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

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

  ```
  Stack<int> // template class
  ```
- A template function explicitly declared:

  ```
  void print(int); // template function
  ```
- A template function implicitly declared by calling it or taking its address:

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

template instantiation

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

thread

sets a thread to be the current one or affects the display of information about a thread

cdm	pdm
	✔

Command syntax

thread

thread [*tid***]**

thread -info *tid*

Descriptiont

<< *none* >>	Returns the identifier (*tid*) of the current thread.
tid	Sets thread *tid* to be the current thread.
-info *tid*	Gives information about *tid*, or the current thread.

Usage

Use the thread command at a break location to set a thread to be the current one or to affect the display of information about a thread.

When you issue the **thread -info** command to display information about a thread, ObjectCenter displays

- An arrow (->) if the thread is the current thread.
- Thread id (**t@***number*)

  The thread id is the **thread_t** value that **thr_create** passes back.
- Whether it is bound (**b**) or active (**a**)

  If the thread is bound or active, the thread is running on a light-weight process (LWP). A light-weight process is a kernel thread. For running threads, the LWP id also appears (**l@***number*).
- Start function

  The start function is the function the program passed to **thr_create()**. A question mark appears instead of the function name if the function is unknown.

- Thread state

  The state of a bound or active thread is the state of its LWP.

Thread state	The thread ...
running	Is running when the program reaches the breakpoint
runnable	Is runnable
suspended	Is explicitly suspended
zombied	Has exited but has not yet joined the main thread
sleeping	Is blocked
sleep on	Is blocked on *synchron_object*, *synchron_object* where *synchron_object* is the address of the mutex lock or other synchronization object.
unknown	Has a state that ObjectCenter is unable to determine

- The function the thread is currently executing

You control execution of the current thread with the **cont**, **next**, **nexti**, **step**, and **stepi** commands. You can display a traceback of the thread execution stack with the **where** command. The **threads** command displays information about all the threads active at a break location.

Example

The following extract from a sample run of a threaded version of the bounce demo used in the tutorial show the output of the **thread** and **thread -info** commands:

```
pdm (break 1) 6 -> thread
The current thread is t@5
pdm (break 1) 7 -> thread -info t@5
>   t@5   a l@1
thread_bounce__FP13DrawableShape()running in
doDraw__13DrawableShapeFv()
```

The next example shows how you can use the **thread** command to change the current active thread and then use **stepi** to step through machine instructions or **step** to step through execution.

```
pdm (break 1) 19 -> thread
The current thread is t@6
pdm (break 1) 22 -> thread t@5
The current thread is t@5
pdm (break 1) 23 -> stepi
0xef71eff4 <_thrsys_poll+8>:    bcc  0xef71f01c
<_thrsys_poll+48>
pdm (break 1) 25 -> step
Single stepping until exit from function
_thrsys_poll, which has no line number information.
0xef71efb8 in _poll ()
```

You can use the **where** command to examine the execution stack. Here we issue the **where** command with **t@5** the current thread, then use the **thread** command to make **t@6** the current thread and issue the **where** command again:

```
pdm (break 1) 26 -> thread -info t@5
>    t@5   a l@1 thread_bounce__FP13DrawableShape()running in _poll()
pdm (break 1) 27 -> where
#0  0xef71efb8 in _poll ()
#1  0xef67e658 in _select ()
#2  0x12420 in DrawableShape::wait(void) (this=0x27c10) at shapes.C:149
#3  0x123a4 in DrawableShape::drawMove(int) (this=0x27c10, count=0) at
shapes.C:127
#4  0x122e4 in DrawableShape::doDraw(void) (this=0x27c10) at shapes.C:113
#5  0x123d0 in DrawableShape::bounce(void) (this=0x27c10) at shapes.C:132
#6  0x11b50 in thread_bounce(DrawableShape *) (shape=0x27c10) at
mainTh.C:12
pdm (break 1) 28 -> thread t@6
The current thread is t@6
pdm (break 1) 29 -> thread -info t@6
 >   t@6   a l@4 thread_bounce__FP13DrawableShape()running in
doDraw__13DrawableShapeFv()
pdm (break 1) 30 -> where
#0  DrawableShape::doDraw(void) (this=0x27c48) at shapes.C:110
#1  0x123d0 in DrawableShape::bounce(void) (this=0x27c48) at shapes.C:132
#2  0x11b50 in thread_bounce(DrawableShape *) (shape=0x27c48) at
mainTh.C:12
pdm (break 1) 30 ->
```

Restrictions

This command is unavailable on some platforms. Refer to "Product limitations" in the online "About This Release" document.

See Also

cont, **next**, **nexti**, **step**, **stepi**, **thread support**, **threads**, **where**

thread support

Thread support on Solaris 2

We've added support for threaded applications on the Solaris 2 platform in process debugging mode (pdm), with the ability to debug threads in executables and a graphical Thread Browser to show the status of all the threads in your program. We have also added a thread-safe **libC** (the C++ library). Use the **-mt** switch to CC to compile and link multi-threaded programs.

In process debugging mode, the Thread Browser gives you information about the threads and lightweight processes in your program. This information includes a list of all threads, and the state of each thread. The state information includes the function the thread is executing, the execution state (for example, running, sleeping) of the thread, and the start function for the thread.

At any given time, the Thread Browser focuses on a single thread or light-weight process (LWP), known as the "current active entity." You control execution of the current thread with the **cont**, **next**, **nexti**, **step**, and **stepi** commands. You can display a traceback of the thread execution stack with the **where** command. You can also perform these operations on another thread at the break level by making it the current active entity. To make another thread the current active thread, you use the **thread** command with the new thread number as an argument.

See Also **CC**, **thread**, **threads**

threads

displays information about threads active at a break location

cdm	pdm
	✔

Command syntax **threads**

Description

<< *none* >>	Lists in the Workspace information about the state of threads at a break location

Usage

Use the **threads** command at a break location to examine information about active threads. All active threads stop when execution reaches a breakpoint.

When you issue the **threads** command ObjectCenter displays

- On the first line, the process id and the name of the process you are debugging.
- On each subsequent line, information about an active thread.

The thread line with an arrow (>) is the current thread, referred to as the current active entity.

Each thread line contains the following information.

- Thread id (**t@***number*)

  The thread id is the **thread_t** value that **thr_create** passes back.

- Whether it is bound (**b**) or active (**a**)

  If the thread is bound or active, the thread is running on a light-weight process (LWP). A light-weight process is a kernel thread. For running threads, the LWP id also appears (**l@***number*).

- Start function

  The start function is the function the program passed to **thr_create()**. A question mark appears instead of the function name if the function is unknown.

- Thread state

  The state of a bound or active thread is the state of its LWP.

Thread state	**The thread ...**
running	Is running when the program reaches the breakpoint
runnable	Is runnable
suspended	Is explicitly suspended
zombied	Has exited but has not yet joined the main thread
sleeping	Is blocked
sleep on	Is blocked on *synchron_object*, *synchron_object* where *synchron_object* is the address of the mutex lock or other synchronization object.
unknown	Has a state that ObjectCenter is unable to determine

- The function the thread is currently executing

You control execution of the current thread with the **cont**, **next**, **nexti**, **step**, and **stepi** commands. You can display a traceback of the thread execution stack with the **where** command. To change the context to another thread or display information about an individual thread, use the **thread** command. You can also change the focus to another thread by clicking the line in the Thread Browser that shows the thread you want to follow.

Example

The following example shows sample output of the **threads** command from a threaded version of the bounce program used in the tutorial. The **threads** command is issued before and after execution steps through a call to the **DrawableShape** function. Notice that the **>** indicates that **t@1** is the current active entity before the call, and **t@5** is the current active entity after the call.

```
pdm (break 1) 222 -> threads
Proc 3141
Thread  LWP Start          State               Where
t@1   a l@1 0x0()          running             in  main()
t@2         0x0()          sleep on 0xef738460  in _swtch()
t@3   b l@2 0x0()          running             in  sigwait()
t@4   b l@3 _co_timerset() running             in  lwp_sema_wait()
t@5 thread_bounce__FP13DrawableShape()runnable  in _setpsr()
t@6 thread_bounce__FP13DrawableShape()runnable  in _setpsr()
pdm (break 1) 223 -> step
pdm (break 1) 224 -> threads
Proc 3141
Thread LWP Start           State                 Where
t@1         0x0()          sleep on 0xef730b08   in _swtch()
t@2         0x0()          sleep on 0xef738460   in _swtch()
t@3   b l@2 0x0()          running               in _sigwait()
t@4   b l@3 _co_timerset() running               in _lwp_sema_wait()
t@5   a l@1 thread_bounce__FP13DrawableShape()running  in
doDraw__13DrawableShapeFv()
t@6         thread_bounce__FP13DrawableShape()runnable in setpsr()
```

Restrictions

This command is unavailable on some platforms. Refer to "Product limitations" in the online "About This Release" document.

See Also

cont, **next**, **nexti**, **step**, **stepi**, **thread**, **thread support**, **where**

touch

marks memory as initialized and valid

cdm	pdm
✔	

Command syntax

touch *address*
touch *lvalue*
touch *size* **at** *address*
touch *size* **at** *lvalue*
touch *size* **at** *variable*
touch *variable*

Description

address	Marks the data space at *address* as initialized and valid. *The address* argument must be a hexadecimal value.
lvalue	Evaluates *lvalue*, treats the resulting value as an address, and marks the data space at that address as initialized and valid.
size **at** *address*	Marks the specified number of bytes (*size*) of data space at *address* as initialized and valid. The *address* argument must be a hexadecimal value.
size **at** *lvalue*	Evaluates *lvalue*, treats the resulting value as an address, and marks the specified number of bytes (*size*) of data space at that address as initialized and valid.
size **at** *variable*	Marks the specified number of bytes (*size*) of data space for *variable* as initialized and valid.
variable	Marks the data space of *variable* as initialized and valid.

Options

The following ObjectCenter option affects the **touch** command:

save_memory When **save_memory** is set, you do not need to use **touch** where you otherwise would.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **touch** command to mark memory as initialized and valid in order to prevent warnings about garbage values and type mismatches when the memory is used.

If *size* is not specified, the size of the type of the expression is used. ObjectCenter uses one byte for an *address* argument.

In the rare case that memory is allocated in your program's address space without ObjectCenter's knowledge, use the **touch** command to inform ObjectCenter that the memory exists. For example, unless you issue the **touch** command, ObjectCenter cannot determine the memory allocation for an undocumented system call or other local operating system modification.

Example

The example below shows how **touch** can be used to suppress type mismatch warnings. A value is stored as a **char**, but examined as an **int**.

```
-> int *ptr;
-> ptr=malloc(16);
Warning #608: Questionable argument type:
 (int *) = (int)
(int *) 0x195f48 /* (allocated) */
-> *(char *)ptr =0;
(char) '\000'
-> *ptr;
Warning #112: Retrieving a <int> from allocated data
at <0x195f48>.
The object stored there is a <char>.
(break 1) -> cont
(int) 12566463
-> touch *ptr
-> *ptr;
(int) 12566463
->
```

Note that this example touches the allocated memory, not the variable **ptr** itself. If you want to touch the variable **ptr**, the command would be:

```
-> touch ptr
```

Sometimes you will want to embed calls to the **touch** command within a function. To do that, use the **centerline_untype()** function.

Restrictions

ObjectCenter initializes allocated data and local variables to the value **191** in order to perform *used before set* checks. It is possible for spurious warnings to occur if the value **191** is stored in this memory while the program is executing within object code. Source code that uses **191** as a legitimate value will *not* generate spurious warnings.

Touching this memory will eliminate these spurious warnings. The warnings can also be suppressed with the **suppress** command, and the default value can be changed by modifying the **unset_value** option.

See Also

setopt, **suppress**, **centerline_untype()**

trace

traces program execution

cdm	pdm
✔	

Command syntax

trace

trace *function*

Description

<< *none* >>	Displays each line of source code as it is being executed.
function	Displays each line of source code in the specified function as it is being executed.

Usage

Use the **trace** command to display each line of source code as it is being executed. If a function is specified, tracing is limited to that function.

Statements executed within an **action** are not traced.

To turn off tracing, use the **delete** command.

Restrictions

You cannot use **trace** within object code in component debugging mode.

See Also

delete, **next**, **step**, **stop**, **status**

unalias

removes an alias for a command

cdm	pdm
✔	✔

Command syntax **unalias** *name*

Description

name Deletes the the alias specified by *name.*

Usage

Use the **unalias** command to delete an alias that you no longer want to use.

Example

If you have an alias named **p** that invokes the **print** command, you can delete the alias with the following command:

```
-> unalias p
```

See Also **alias**

uninstrument

disables run-time error checking for an object file

cdm	pdm
✔	

Command syntax

uninstrument

uninstrument *file* **...**

uninstrument all

Description

<< *none* >>	Prompts you to remove instrument information from files one at a time.
file ...	Removes instrumentation information from *file.*
all	Removes instrumentation information from all files.

Usage

Use the **instrument** and **uninstrument** commands to enable and disable run-time error checking of loaded object code. Enabling the run-time error checking of loaded object code is called *instrumenting* the file.

Performance considerations

When you disable run-time error checking for a particular module, that module runs somewhat faster than an object module with run-time error checking enabled. See 'Run-time error checking in source or object code' on page 164 and 'Loading source versus object code versus executables' on page 116 for more information about the performance trade-offs.

See the **instrument** entry on page 162 for more information about instrumenting and uninstrumenting.

See Also

debugging, **instrument**

unload

unloads files

cdm	pdm
✔	

Command syntax

unload

unload all

unload *file ...*

unload *function*

unload *library*

unload *library(module)*

unload templates

unload user

unload workspace

Description

<< *none* >>	Prompts for unloading files one at a time.
all	Unloads all files, including all libraries and all modules linked from libraries.
file ...	Unloads the specified files. Takes shell wildcards so you can unload groups of files with one command.
function	Unloads the file containing the specified function. If the specified function is linked from a static library, unloads only the object module containing the function, not the entire library.
library	Detaches the entire specified library and unloads the individual modules that have been linked in from that library.

library(module)	Unloads the specified module in the specified library. For example: -> **`unload /lib/libc.a(printf.o)`**
templates	Unloads all template instantiation modules. See the **templates** entry on page 335information about templates.
user	Unloads all source and object code files currently loaded.
workspace	Unloads all definitions entered in the Workspace. Files remain loaded, and libraries remain attached.

Switches

-l*x*	Unloads the specified library **lib***x***.a**.

Usage

Use the **unload** command to unload files from ObjectCenter.

Because **unload** automatically resets to the top level of the Workspace before unloading a file, you cannot unload a file at a break level and then continue execution.

All functions and static variables defined by the unloaded file become undefined. Global data variables, macro definitions, classes, structures, unions, enumerators, enumeration constants, and type definitions will not become undefined if they are declared or defined in any other loaded file.

Breakpoints, watchpoints, traces, and actions are deleted if they are set on variables or functions that become undefined when a file is unloaded.

When specifying a file as an argument for **unload**, you can use shell wildcards to unload groups of files with one command. For example:

```
-> load str_1.C str_2.C str_3.C main.C
-> unload str*.C
Unloading: str_1.C
Unloading: str_2.C
Unloading: str_3.C
```

To unload an individual library module, that module must be linked in.

See Also

build, **contents**, **load**, **make**, **swap**, **templates**

unres

lists undefined variables and functions

cdm	pdm
✔	

Command syntax

unres

unres *function*

unres *variable*

Description

<< *none* >>	Lists all undefined variables and functions that are referenced by the program.
function	Lists the undefined variables and functions referenced by the specified function.
variable	Lists the undefined variables and functions used as initialization values for the specified variable.

Usage

Use the **unres** command to list undefined variables and functions that are referenced by your program.

See Also

link, **load, unload, xref**

unsetenv

removes a variable from the program's environment

cdm	pdm
✔	✔

Command syntax

unsetenv *variable*

Description

variable Removes the definition of *variable* from the system environment.

Usage

Use the **unsetenv** command to remove a variable from the program's system environment. The **unsetenv** command is analogous to the similarly named shell command.

The **unsetenv** command affects only your program's environment variables. It does not affect the environment variables used by ObjectCenter to control its own operations.

The environment is an array of strings that is made available to the program through the global **environ** variable and the **envp** parameter, which is passed as the third argument to the **main()** function. By convention, each string has the format *name*=*value*, where the *value* part is optional.

Warnings

If **unsetenv** is called from a break level, it will alter the value of the global **environ** variable, but not the **envp** parameter passed to **main()**. This problem also occurs with the **putenv()** function.

Changing the **EDITOR** or **DISPLAY** shell variables with **unsetenv** does not affect which editor or display screen ObjectCenter uses.

See Also

printenv, **setenv**

unsetopt

unsets an ObjectCenter option

cdm	pdm
✔	✔

Command syntax

unsetopt

unsetopt *option*

Description

<< *none* >>	Displays all options that are unset.
option	Unsets the specified option as follows:
	If the option takes a string, the option is assigned the empty string.
	If the option takes an integer, the option is assigned 0.
	If the option takes a Boolean, the option is assigned FALSE.

Usage

Use the **unsetopt** command to examine and change ObjectCenter options. See the **options** entry for more details about each option.

See Also

printopt, **setopt**

unsuppress

reactivates reporting of a warning

cdm	pdm
✔	

Command syntax

unsuppress
unsuppress *num*
unsuppress *num* **everywhere**
unsuppress *num* **[at]** *line*
unsuppress *num* **[at]** *"file":line*
unsuppress *num* **[in]** *directory*
unsuppress *num* **[in]** *file*
unsuppress *num* **in** *function*
unsuppress *num* **[in]** *lib***(***module***)**
unsuppress *num* **[on]** *identifier*
unsuppress *num* **[on]** *function*

Description

<< *none* >>	Prompts you to reactivate suppressed warnings one at a time.
num	Reactivates reporting of each occurrence of the specified warning, regardless of whether you had suppressed it globally or with a location-specific argument.
num **everywhere**	Reactivates reporting of a warning that you had suppressed globally (without a location-specific argument).
num **[at]** *line*	Reactivates reporting of the specified warning at the specified line
num **[at]** *"file":line*	Reactivates reporting of the specified warning at the specified line in the specified file.

num **[in]** *directory*	Reactivates reporting of the specified warning in all files in the specified directory or in any subdirectories of the specified directory.
num **[in]** *file*	Reactivates reporting of the specified warning in the specified file.
num **in** *function*	Reactivates reporting of the specified warning while in the specified function.
num **[in]** *lib***(***module***)**	Reactivates reporting of the specified warning in the specified module of the specified library.
num **[on]** *identifier*	Reactivates reporting of the specified warning if the warning involves the specified identifier. The *identifier* argument is any variable, typedef, class / struct / union tag, or macro name.
num **[on]** *function*	Reactivates reporting of the specified warning when the specified function is called.

Usage

Use the **unsuppress** command to reactivate the reporting of warnings and errors, known collectively as violations.

Handling lint comments

If the comment /*SUPPRESS *n*/ appears in source code, static error checking is suppressed for the specified violation. If the comment appears at the global level of a file, the violation is suppressed for the entire file. If the comment appears within a function, the violation is suppressed only for the following line.

See Also

suppress

up

moves up the execution stack

cdm	pdm
✔	✔

Command syntax

up

up *number*

Description

<< *none* >>	Moves the current scope location up one level on the execution stack. Motif or OPEN LOOK:The Source area shows file scoped to location and highlights it with an arrow.
number	Moves the current scope location the specified number of levels up the execution stack.

Usage

Use the **up** command to move the current scope location up the execution stack, toward the top level of the Workspace and away from the current break level.

The scope location is the point at which all variables, types, and macros are scoped. When a break level is generated, the scope location is set to the point at which execution was interrupted.

When at a break level, use the **where** command to display the execution stack. Use the **whereami** command to display the break location and the current scope location.

The **cont** command can be used to continue execution, and the **reset** command can be used to return to a previous break level or to the top level of the Workspace without continuing execution.

See Also

cont, **down**, **reset**, **where**, **whereami**

use

displays or sets the directory search path

cdm	pdm
✔	✔

Command syntax

use

use *pathname ...*

Description

<< *none* >>	Displays the current directory search path.
pathname ...	Sets the list of directories to be searched to the specified pathname. If more than one pathname is listed, they must be separated by spaces. In process debugging mode they may be separated by spaces or colons. The directories can be specified as absolute or relative pathnames.

Options

The following ObjectCenter options affect the **use** command:

cxx_suffixes	Specifies file extensions to search for when ObjectCenter needs to find a C++ source file that corresponds to a given object file.
path	Specifies the search path for loading source and object files (not for **#include** files). You must also set the **swap_uses_path** option for the **path** option to affect the **swap** command.
swap_uses_path	Determines whether the **swap** command uses the path, which can be set by the **use** command or by the **path** option.

See the **options** entry for more details about each option. ObjectCenter does not support these options in process debugging mode (**pdm**).

Usage

Use the **use** command to set the list of directories to be searched when a filename is given to the **debug, edit, list, load**, or **swap** commands. The **swap** command uses the path set by **use** only if the **swap_uses_path** option is set.

In component debugging mode, the **use** command sets and displays the current value of the **path** option, which can also be set and displayed with the **setopt** and **printopt** commands, respectively.

Restrictions

The **use** command does not provide a search path for loading **#include** files, only for loading source and object files.

To give the search path for **#include** directories, use the **-I** switch with the **load** command according to the following format:

```
load -Iinclude_dir1 [-Iinclude_dir2 ...] file...
```

See Also

cd, **debug**, **edit**, **list**, **load**, **printopt**, **setopt**, **swap**

user-defined commands

ObjectCenter allows you to define commands for the graphical user interface. See the **X resources** entry on page 436.

whatis

lists all uses of a name

cdm	pdm
✔	✔

Command syntax **whatis** *name*

Description

name Displays all uses of the specified name as a function, variable, class / struct / union tag name, enumerator, type definition, or macro definition.

Usage

Use the **whatis** command to display all uses of an identifier name. An identifier name is a name for a function, variable, enumerator, class / struct / union tag name, type definition, or macro definition.

ObjectCenter first displays all uses of the name within scope at the current scope location, followed by all uses of the name not within scope. The order of the listing represents the order in which the specified name is resolved when it is used.

Example

In the following example, the name **test** is used as both a variable and a macro.

```
 -> int test;
 -> #define test 100
 -> whatis test
#define test 100
extern int test; /* initialized */
 -> int test2=2*test;
 -> test2;
(int) 200
```

Because you cannot declare or define variables in the Workspace in pdm, the preceding example works only in cdm.

See Also **dump**, **display**, **expand**, **help**, **list**, **man**, **print**, **whereis**, **xref**

when

executes specified commands

cdm	pdm
	✔

Command syntax

when

when if *cond*

when [at] *line*

when [at] *line* **if** *cond*

when in *func*

when in *func* **if** *cond*

Description

<< *none* >>	Executes commands at current location.
if *cond*	Executes commands at current location if *cond* is true, where *cond* is a Boolean expression.
[at] *line*	Executes commands when the specified line in the current file is reached.
[at] *line* **if** *cond*	Executes commands when the specified line in the current file is reached if *cond* is true, where *cond* is a Boolean expression.
in *func*	Executes commands at the first line in the specified function.
in *func* **if** *cond*	Executes commands at the first line in the specified function if *cond* is true, where *cond* is a Boolean expression.

Usage

Use the **when** command to set debugging actions in pdm; use the **action** command in cdm. The two commands are very similar.

After you issue the **when** command, ObjectCenter prompts you for the commands to be executed. These commands can include calls to functions that are defined in the program.

By default, ObjectCenter remains stopped after executing the commands specified with **when**. If you want your program to continue after executing the commands, you must specify the **cont** command as the last one.

Example

Here is an example of how to use the **when** command:

```
(pdm)  4 -> when at 5 if i == 100
```

Then type commands to be executed (one per line). Typing "." or "end" completes the sequence.

```
when -> printf("in func : %d\n", i);
when -> i = 200;
when -> cont
when -> .
(pdm)  5 ->
```

See Also

action

where

displays the execution stack

cdm	pdm
✔	✔

Command syntax

where

where *number*

Description

<< *none* >>	Displays a traceback of the execution stack, starting from the location where the execution has stopped.
number	Displays a traceback of only the specified number of functions on the top of the execution stack. The most recent routines called are at the top of the stack.

Options

The following ObjectCenter option affects the **where** command:

terse_where	Tells the **where** command not to list the formal arguments of each function on the execution stack.

See the **options** entry for more details about each option. ObjectCenter does not support this option in process debugging mode (**pdm**).

Usage

Use the **where** command to display a traceback of the execution stack.

When execution is stopped in object code, it is often useful to see a full stack trace with arguments. The **where** command displays the formal parameters of source code functions and of object code functions that contain debugging information.

In component debugging mode but not process debugging mode, the formal parameters of object code functions without debugging information can be displayed by entering the prototype for the function. After a prototype is entered into the Workspace, all subsequent stack traces will show the arguments to that function. For more information on using prototypes, see the **proto** entry on page 292.

In threaded applications, **where** shows the stack trace for the current active thread. To see the stack trace for a different thread, issue the **thread** command with the identifier of the other thread as the argument.

NOTE Debugging of threaded applications is currently only supported in process debugging mode, and it is not supported on all platforms. Please refer to "Product limitations" in the online "About This Release" document for more information.

Example

This example shows the output of the **where** command for three different threads in a threaded version of the bounce demo used in the ObjectCenter tutorial. The **>** character in the output of the **threads** command shows that **t@5** is the current active entity.

```
pdm (break 1) 18 -> threads
Proc 24910
Thread  LWP Start             State               Where
  t@1       0x0()             sleep on 0xef730b08 in _swtch()
  t@2       0x0()             sleep on 0xef738460 in _swtch()
  t@3 b l@2 0x0()             running             in __sigwait()
  t@4 b l@3 _co_timerset()    running             in __lwp_sema_wait()
> t@5 a l@1 thread_bounce__FP13DrawableShape()running in getX__5PointFv()
  t@6       thread_bounce__FP13DrawableShape()runnable in _setpsr()
pdm (break 1) 19 -> where
 #0 Point::getX(void) (this=0x27c24) at shapes.C:52
 #1 0x12444 in DrawableShape::createTable(void) (this=0x27c10) at
shapes.C:164
 #2 0x122c0 in DrawableShape::doDraw(void) (this=0x27c10) at shapes.C:110
 #3 0x123d0 in DrawableShape::bounce(void) (this=0x27c10) at shapes.C:132
 #4 0x11b50 in thread_bounce(DrawableShape *) (shape=0x27c10) at
mainTh.C:12
pdm (break 1) 20 -> thread t@6
The current thread is t@6
pdm (break 1) 21 -> where
 #0  0xef71d544 in _setpsr ()
 #1  0x11b44 in thread_bounce(DrawableShape *) (shape=0x0) at mainTh.C:11
pdm (break 1) 22 -> thread t@1
The current thread is t@1
pdm (break 1) 23 -> where
 #0  0xef716848 in _swtch ()
 #1  0xef71a664 in _thr_join ()
 #2  0xef7192b8 in _reap_wait ()
 #3  0xef71a664 in _thr_join ()
 #4  0x11c6c in main () at mainTh.C:34
pdm (break 1) 24 ->
```

See Also

cont, **down**, **options**, **proto**, **up**, **whereami**

whereami

displays the current break and scope locations

cdm	pdm
✔	✔

Command syntax

whereami

Description

<< *none* >> Displays the current break and scope locations.

Usage

Use the **whereami** command to list the current break location and the current scope location. This is particularly useful for finding where you are once you have moved up or down the execution stack while at a break level.

Break location

The break location is the point at which execution stopped when the break level was entered.

Scope location

The scope location is the point to which variables, functions, and types are scoped. When a break level is entered, it is set to the break location. It can be changed to different locations on the execution stack with the **up** and **down** commands.

Display of locations

If you have not moved up or down in the execution stack while at a break level, the scope location and the break location are the same. The **whereami** command displays that location in the Source area, scrolling the display if necessary.

If you have moved up or down in the execution stack, the scope location is displayed in the Source area and the break location is shown in the Workspace.

NOTE If the **whereami** command appears not to respond as you expect, keep the following in mind:

- The break location is only displayed in the Workspace when the break location is different from the scope location.
- The Source area will only change if the current scope location is not already displayed there.

Display of locations in Ascii ObjectCenter

In Ascii ObjectCenter, if you have not moved up or down in the execution stack while at a break level, the break location and the scope location are the same. In this case, **whereami** gives a single listing for both the break and the scope locations.

Errors and warnings

If an error or a warning caused the current break level, **whereami** displays the error or warning number. If execution can be continued from the break level, **whereami** displays the arguments that can be passed to the **cont** command.

See Also

cont, **down**, **proto**, **up**, **where**

whereis

lists the locations where a name is declared or defined

cdm	pdm
✔	✔

Command syntax **whereis** *name*

Description

name Lists the locations where a name is declared or defined; lists only global and top-level static declarations.

Usage Use the **whereis** command to list locations where a symbol is declared or defined as a global or top-level static.

Example In the following example, the name **test** is used as both a variable and a macro.

```
-> int test;
-> #define test 100
-> whereis test
"workspace":2 #define test 100
"workspace":1 int test, defined
```

NOTE Because you cannot declare or define variables in the Workspace in pdm, the preceding example works only in cdm.

See Also **list**, **display**, **whatis**, **xref**

window managers

In general, we recommend that you use **mwm, olwm**, or **olvwm** as your window manager when you are running ObjectCenter. Although you may be able to use other ICCCM-compliant window managers with some success, ObjectCenter does not support other window managers explicitly.

Actions used to "Quit"

Some window managers provide two different functions for getting rid of a window: **f.delete** and **f.destroy**. We recommend that you be careful about distinguishing them.

The **f.delete** function sends an ICCCM WM_DELETE_WINDOW message to the selected window, which causes the window to disappear. The **f.destroy** function tells the X11 server to sever the connection to the client owning the selected window, killing that client process as a result.

Be careful about binding **f.destroy** to a menu item like "Quit". If you do so, and your application is a multi-window application, the whole application will die. This problem can also result from binding **f.destroy** to a "Quit" item in a dialog box or "pinned" menu. In this case, you should probably use **f.delete** instead of **f.destroy.**

Bringing transient windows to the front

All of ObjectCenter's non-top-level windows, such as the dialog boxes and pinnable property sheets, are transient for the top-level window to which they belong.

As of OpenWindows 3.0, by default, **olwm** forces all transients for a given window to appear stacked above that window. If you try to raise the "parent" window, all the transients are raised too.

This means, for example, that if you request the Contents window from the Project Browser window, you cannot bring the Project Browser window to the front.

If you want to change this behavior, put the following line in your **.Xdefaults** file:

```
OpenWindows.KeepTransientsAbove: False
```

Workspace

All versions of ObjectCenter provide an interactive work area, called the Workspace, that handles ObjectCenter commands and C++ statements. See the *User's Guide* for basic information about the Workspace.

Here we describe more advanced ObjectCenter features that help you enter input into the Workspace. We cover the following topics:

- Saving a transcript of your session
- Displaying your input history
- Saving your input history
- Repeating previous input
- Expanding variables in the Workspace
- Using shell meta-characters and operators
- Line editing
- Using name completion
- Redirecting output
- Specifying a variable's location
- Changing and listing directories
- Entering C++ code in the Workspace
- Unloading the Workspace scratchpad
- Clearing the Workspace
- Using the **edit workspace** command

Saving a transcript of your session

At any point during an ObjectCenter session, you can save a transcript of your Workspace actions in a file. If you do so, the transcript contains all of your input as well as all of ObjectCenter's output.

To save a transcript, open the Workspace pop-up menu, and select **Save to.** You can then either select the default name, which is **~/ocenter.script,** or specify a different name by selecting **Other file**.

Displaying your input history

Use ObjectCenter's **history** command to display previous input:

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

If you are using the Motif or OPEN LOOK version, you can also display your previous input in the Workspace by using the mouse-based scrollbar on the right side of the Workspace panel.

The Workspace features a history mechanism modeled after the **csh** and **tcsh** shells. Similar to the **tcsh** shell, previous lines of input can be scanned one line at a time by entering Control-p to scan backward and Control-n to scan forward.

```
-> int i;
-> double d;
-> char c;
-> <Ctrl-p>< Ctrl-p>       expands to...
-> double d;
```

You can also type a letter or letters, then press Control-p or Control-n to scan through command lines that began with those letters. For example, typing **load** then pressing Control-p repeatedly scans all the previous lines that loaded files.

Saving your input history

ObjectCenter saves all Workspace input in a temporary logfile that it deletes at the end of a session. You can specify the name of the logfile with the **logfile** option.

You can tell ObjectCenter to keep a permanent logfile by using the **-f** command-line switch when starting ObjectCenter. For more information, see the "Switches" section of the **objectcenter** entry on page 221.

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

```
-> history #> ocenter_log_name
```

The logfile records input only; it does not show ObjectCenter's output. If you are using Motif or OPEN LOOK, you can record output in a file as well, by opening the Workspace pop-up menu and selecting **Save to**.

You should never edit or modify the logfile during a session; ObjectCenter uses it just for recording input. Changing the logfile does not affect the state of the Workspace. If you want to modify the Workspace along with the logfile, it is best to copy the logfile to another file, edit it, unload the Workspace, and load the edited copy of the logfile with the **source** command. See the **source** entry on page 316 for more information.

Repeating previous input

As in the **csh** shell, you can execute previous lines of input using a history character followed by an argument. ObjectCenter's history character is # (the **csh** shell's history character is !).

Repeating the most recent line of input

You can repeat the most recent line of input by entering **##**<*return*>, where <*return*> represents the Return key. Alternatively, you can enter **##**<*space*> to edit the line before it is re-entered:

```
-> (123 + 456);
(int) 579
-> ##<space>              expands to...
-> (123 + 456);
(int) 579
```

Revising the most recent line of input

Entering #^*old_string*^*new_string* redoes the most recent line of input, with the string *new_string* substituted for the string *old_string*:

```
-> (123 + 456);
(int) 579
-> #^123^333<space>       expands to...
-> (333 + 456);
(int) 789
->
```

Repeating a particular line of input

You can repeat any line of input using the notation #*text*, where *text* matches the beginning of a previous line of input. You can also repeat a previous line of input using the notation #*n*, where *n* is the number of the input line to be redone, or #-*n*, where *n* is the *n*th previous command.

For example:

```
-> (123 + 456);
(int) 579
-> (321 + 654);
(int) 975
->
-> #( <space>                expands to...
-> (321 + 654);
(int) 975
->
-> #1 <space>                expands to...
-> (123 + 456);
(int) 579
->
-> #-3 <space>               expands to...
-> (321 + 654);
(int) 975
->
```

Expanding particular tokens of previous input

You can expand selected tokens of the previous line of input, similar to using the **csh** shell's history commands **!$**, **!***, and **!:**. See Table 31 for the syntax.

Table 31 Syntax for Expansion of Tokens in Workspace Input

Symbol	Expansion
#$<*space*>	Last token of the previous line of input
#*<*space*>	All but the first token of the previous line of input
#: *number*<*space*>	The *number* token on the previous line (tokens are numbered starting at **0**)

Here is an example:

```
-> int i, j, k;
-> i = 123 + 456;
(int) 579
-> j #* <space>                  expands to...
-> j = 123 + 456;
(int) 579
-> k = #:4 <space>               expands to...
-> k = 456 ;
(int) 456
```

Expanding variables in the Workspace

You can expand the value of any environment variable or ObjectCenter option in any Workspace command by using the **#$** syntax shown in Table 32.

Table 32 Syntax for Expansion of Environment Variables and Options in Workspace Commands

Symbol	Expansion
#$ *identifier*	Substitutes the value of the ObjectCenter option, if one exists, named *identifier*; otherwise, substitutes the value of the named environment variable. For example, **#$path** substitutes the value of the ObjectCenter **path** option, if it is set; **#$HOME** substitutes the current value of the **HOME** environment variable.
#$*(environ_var)*	Substitutes the value of the named environment variable. For example, including **#$(HOME)** substitutes the current value of the **HOME** environment variable. Note that text must be enclosed in parentheses **()**.
#$*{option}*	Substitutes the named ObjectCenter option value. For example, **#${load_flags}** substitutes the loading flags that you have set in ObjectCenter. Note that text must be enclosed in braces **{ }.**

In the following examples, **#$HOME** is expanded to the value of the **HOME** environment variable.

```
-> printenv HOME
HOME=/s/users/jk
-> load #$HOME/sample.c
Loading: /s/users/jk/sample.c
```


In the following example, a directory is added to ObjectCenter's search path by expanding **#${path}** to the current value of ObjectCenter's **path** option, then specifying the directory to add to the path.

```
-> printopt path
path     /s3/bobh/src
-> setopt path #${path} /s3/bobh/obj
-> printopt path
path     /s3/bobh/src /s3/bobh/obj
```

Viewing the expansion before the command is executed

You can see what the arguments expand to before executing a command by pressing the Spacebar instead of pressing Return.

Using shell meta-characters and operators

The following commands support the shell operators and meta-character expansion supported by **/bin/sh**:

- **cd**
- **ls**
- **load**
- **make**
- **sh**
- **shell**

All commands except the **unload** command attempt to match the expanded wildcard name to the list of files on the disk. The **unload** command tries to match against the loaded filenames.

For example, you can issue the following command in the Workspace:

```
-> load *.c
Loading: bad_ptr.c
Loading: clean_att.c
Loading: cleanfile.c
Loading: copyfile.c
```

Line editing

The Workspace supports line editing of input similar to the line editing available in the **emacs** editor. See Table 33 for the most frequently used line-editing commands. Note that not all keyboards have arrow keys.

Table 33 Frequently Used Line-Editing Commands in ObjectCenter

Key	Action
Control-a	Moves the cursor to the beginning of the line
Control-e	Moves the cursor to the end of the line
Control-f	Moves the cursor forward one character
Control-b	Moves the cursor backward one character
Control-d	Deletes the character under the cursor
Up arrow	Scrolls backward through input
Down arrow	Scrolls forward through input
Right arrow	Moves cursor forward one character
Left arrow	Moves cursor backward one character

For a complete list of the default keyboard bindings for ObjectCenter, see the **keybind** entry **keybind** entry on page 167.

Using name completion

The Workspace provides name completion for commands, names, and filename patterns.

You complete commands and names by pressing the Escape key twice without entering text. ObjectCenter handles name completion as follows:

- If ObjectCenter cannot complete the name, it sounds the bell.
- If the completion is ambiguous, the unambiguous portion is completed and all possible matches are listed. For example:

```
-> int ABC, VAR_1, VAR_2;
->
```

```
-> A<ESC><ESC>        completes to...
-> ABC
+> ;
(int) 0
-> V<ESC><ESC>        ambiguous, completes to...
VAR_1       VAR_2
-> VAR_
```

You complete filename patterns by entering the sequence **Esc-x**. This sequence echoes the current input line to the shell specified by ObjectCenter's **subshell** option; the default shell is **/bin/sh**. The subshell echoes the line, performing all filename pattern expansions in the process. For example:

```
-> ls
a.C b.C c.C
-> load *.C<ESC>x     expands to...
-> load a.C b.C c.C
```

Redirecting output

Just as you can redirect output of commands at the shell, you can redirect the output of most ObjectCenter Workspace commands with the following symbols:

Symbol	Result
#> *file*	Redirects the command output to *file.* Overwrites *file* if it exists.
#>> *file*	Appends the command output to the contents of *file.*

Note that the ObjectCenter redirection symbols start with #. For example:

```
-> printenv #> my_vars
```

saves the current environment variables in the file **my_vars.**

NOTE You cannot use the #> syntax to redirect the output of the following commands: **run**, **step**, **next**, **cont**, **reset**.

To redirect the output of the **run** command, use the usual shell syntax for redirection, for example:

```
-> run > stdout_file    (redirect stdout only)
-> run >& output_file   (redirect stdout and stderr)
```

Use the shell syntax to redirect the output of shell commands, such as **ls**, for example:

```
-> ls > listing.output
```

Specifying a variable's location

In certain situations, you must specify the location of a variable to avoid ambiguity. This is usually required for symbols of the same name that are defined differently in different files. The location of a variable can be specified in one of four ways:

- **`file`function`variable**
- **`file`line_number`variable**
- **`file`variable**
- **function`variable**

To use this syntax, you must change to ObjectCenter's C mode with the **cmode** command and use the mangled names of the variables and functions. Keep in mind that the variable must be in scope or on the stack.

For example, assume file **i1**.**C** contains the following top-level declaration:

```
int i = 3;
```

and that **i2**.**C** contains the following top-level declaration:

```
static int i = 1;
```

With these files loaded, ObjectCenter reports on both instances of **i**:

```
C++ 5 -> whatis i
static int i;
extern int i; /* initialized */
```

Because there are two variables named **i** in your program, you specify a location when printing either value, in this case the name of the file:

```
C++ 6 -> cmode
C Workspace Enabled.
C 7 -> whatis i
static int i;
extern int i; /* initialized */
C 8 -> `i1.C`i;
(int) 3
C 9 -> `i2.C`i;
(int) 1
C 10 -> cxxmode
C++ Workspace Enabled.
C++ 11 ->
```

Here's another example in which the variable **i** is defined in two functions in the same file, **test.C**. Use the name of the function to specify the variable's location:

```
C++ 11 -> load test.C
Loading (C++): test.C
C++ 12 -> stop in func
stop (1) set at "test.C":18, func(void).
C++ 13 -> run
Executing: a.out
C++ (break 1) 14 -> step
C++ (break 1) 15 -> whatis i
auto int i; /* Defined in 'func(void)'; currently
active. */
auto int i; /* Defined in 'main'; currently
inactive. */
C++ (break 1) 16 -> cmode
C Workspace Enabled.
C (break 1) 17 -> whatis i
auto int __1i; /* Defined in 'func__Fv'; currently
active. */
auto int __1i; /* Defined in 'main'; currently
inactive. */
C (break 1) 18 -> func__Fv`__1i;
(int) 3
C (break 1) 19 -> main`__1i;
(int) 2
C (break 1) 20 -> cxxmode
C++ Workspace Enabled.
```

Changing and listing directories

The **cd** command changes the current working directory in ObjectCenter in the same manner as the **cd** command in the shell.

Also, you can list the current working directory and list files in the directory by using **pwd** and **ls**, two aliases that ObjectCenter defines automatically to execute Bourne subshells. For example:

```
-> alias
ls            sh ls
pwd           sh pwd
-> pwd
/usr/fenway
-> cd demo
wd now: /usr/fenway/demo
-> ls
Makefile   display.C main.C sort.C sort.h
->
```

Entering C++ code in the Workspace

When you enter C++ code at the Workspace prompt, the Workspace becomes a direct connection to the ObjectCenter interpreter. This means that C++ statements that you type in the Workspace are immediately interpreted, and expressions are immediately evaluated. ObjectCenter displays the result immediately after you type the input.

ObjectCenter maintains a Workspace scratchpad containing C++ definitions, which you can reference throughout a session.

Although you can define functions, variables, and types in the Workspace, it is not easy to edit and modify the definitions. So it is better to put code that needs to be modified or debugged in a source file and then load the source file.

Using multiple-line statements

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

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

NOTE The **+>** prompt only appears when the Workspace is waiting for additional C++ code. This often happens when you forget to type a semicolon (;) at the end of a C++ statement or expression. To complete the input, type a semicolon, then press Return.

Defining variables and types

You can define and use variables and types directly in the Workspace at any time. To display the type and value of a variable, enter the name of the variable followed by a semicolon. (This is a shortcut to using ObjectCenter's **print** command.)

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

```
-> int i;
-> i = 16;
(int) 16
-> while (i<100)i++;
(void)
-> i;
(int) 100
-> int j;
-> j= i+10;
(int) 110
```

For data structures, ObjectCenter displays all non-static members:

```
-> struct mystruct {int i; float f;};
-> struct mystruct struct1;
-> struct1;
(struct mystruct) =
{
 int i = 0;
 float f = 0.000000e+00;
}
```

For pointers, ObjectCenter displays the kind of pointer, the address being pointed to, and the data being pointed to:

```
-> char *msg = "hello there";
-> msg;
(char *) 0x173ea8 "hello there"
```

Defining functions

When defining a function in the Workspace, you must specify its return type and you should specify the types of the parameters when you declare the parameters. For example, to define a function that adds two integers, you could enter:

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

If you don't use a full parameter declaration list, ObjectCenter will tell you that you are using an old-style function definition:

```
-> int add(x,y)
+> int x, y;
+> { return x+y; }
Warning #871: old style definition of add.
```

Calling library functions

You can execute library functions after the library has been attached. For instance:

```
Attaching: /usr/lib/libc.a
-> strlen("hi");
Linking from '/usr/lib/libc.a' ... Linking completed.
(int) 2
```

Declaring types in the Workspace

If an object code file without debugging information is loaded, no information about the types of variables or functions is available.

If the type for a variable defined in compiled code is unavailable, ObjectCenter assigns the generic type **<data>** to the variable. Before variables of type **<data>** can be used in the Workspace, you must declare a type in the Workspace or in a source file.

Consider the file **xyz**.c:

```
int i=4;int test()
{
  return i;
}
```

If this file is loaded into ObjectCenter as an object file compiled with debugging information, you do not need to declare the type of a variable or function that is defined in that file before using it; for example:

```
-> load xyz.o
-> i;
(int) 4
```

However, if this file is compiled without debugging information and loaded into ObjectCenter, this happens:

```
-> load xyz.o
-> i;
Error #739: Variable 'i' has undefined type (<data>).
->
-> extern int i;
-> i;
(int) 4
```

If the type for a function defined in compiled code is unavailable, ObjectCenter assigns the generic type <**text**> to the function.

If the function is called in the Workspace before its type is declared, ObjectCenter gives it the return type **int**:

```
-> load xyz.o
-> test;
Error #739: Function 'test' has undefined type
(<text>).
-> test();
(int) 4
-> test;
(int ()) 0xdc976 < 'test' module "xyz.o" >
```

Responding to errors

ObjectCenter flags errors you make in the Workspace and sets a breakpoint:

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

If you want to return to the top level in the Workspace, issue the **reset** command.

Using blocks

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

For example:

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

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

Using the delete operator in the Workspace

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

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

Unloading the Workspace scratchpad

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

```
-> int add(int x,int y)
+> { return x+y; }
-> add(3,4);
(int) 7
-> unload workspace
Unloading: workspace
-> add(5,6);
Error #733: 'add' is undefined.
```

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

Clearing the Workspace

If you are using the Motif or OPEN LOOK versions, you can clear the Workspace pane by opening the Workspace pop-up menu and selecting **Clear**.

Using the edit workspace command

Use the **edit workspace** command to save code you define in the Workspace. 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 class in the Workspace. You can create stubs for other classes and external functions called by the class, and then invoke the methods in the class to test them. After testing your class, you can use **edit workspace** to create a file containing the code you defined in the Workspace. You can enter your own name for the file or accept the default, **workspace.C**.

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

If you want to test a particular set of definitions, edit the file so that it contains the definitions you want to test. Then use the **unload workspace** command to unload all the definitions and objects you created in the Workspace, and use the **source** command to load the definitions in your saved file back into the Workspace. Note that the **source** command will report errors if you've unloaded any definitions 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. For example, some ObjectCenter commands, such as **whatis**, will appear in the file.

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

---

xref

cross-references a function or variable

cdm	pdm
✔	

Command syntax

xref *function*

xref *variable*

Description

function Lists the functions and variables that reference the specified function and the functions and variables that are referenced by it.

Motif and OPEN LOOK versions: Invokes the Cross-Reference Browser.

variable Lists the functions and variables that reference the specified variable and the functions and variables that are referenced by it.

Usage

Use the **xref** command to cross-reference a specific function or variable.

The **xref** command displays static references that exist in source or compiled code files. A static reference occurs when a function calls another function or uses a global variable. A static reference also occurs when a variable uses the address of a function or variable as an initialization value.

The **xref** command does not display references that are created dynamically during execution. For example, **xref** will not display a reference when the address of a function is assigned to a pointer and the function is subsequently called through the function pointer.

Also, a function's use of its formal parameters or local variables is not displayed.

The compiler removes information about references between functions that are in the same object code file. To avoid the possibility of executing object code that is not fully resolved, ObjectCenter creates implicit references between every function in an object code file. In Ascii ObjectCenter, these references are listed as **implied** by **xref.**

Example

In the following example, **xref** displays a reference between **ptr** and **a**, but not between **ptr** and **b**.

```
int a, b;
int *ptr = &a;
void test()
{
ptr = &b;
}
```

The **xref** command also displays references between the function **test()** and the variables **ptr** and **b**.

See Also

whatis, **whereis**

X resources

Like other X applications, ObjectCenter allows you to customize your Motif or OPEN LOOK Graphical User Interface (GUI) by specifying the values associated with selected resource variables or by using object names to change attributes. You can also use X11 resources to write your own commands.

A complete description of the use of X resources is beyond the scope of this document. In the next few sections, we describe some of the most important resources that you can modify. We have organized this discussion as follows:

- Modifying X resources
- Troubleshooting your .**Xdefaults** file
- Resource descriptions and examples
- Specifying resources for the Run Window and vi Edit window
- Keyboard editing
- Using component and object names
- Examples of changing fonts in particular components
- Examples of using OI (Object Interface Library) components
- Defining GUI-specific resources
- User-defined commands
- Revision control systems
- X resources for the DynaText online documentation browser

Modifying X resources

Like most X11 applications, ObjectCenter reads an **app-defaults** file, which is located as follows:

/ *dir* / **CenterLine** / **oc_2.0.0** / *arch-os* / **lib** / **app-defaults** / **ObjectCenter**

where *dir* is the path to your CenterLine directory and *arch_os* is the platform-specific directory (such as **pa-hpux8** or **sparc-sunos4**).

NOTE Be careful about editing the ...**lib**/**app-defaults**/**ObjectCenter** file. We recommend that you save a copy before you edit so that you do not accidentally lose information about valuable default settings**.**

This is the preferred file to use for modifying Xresources. You can also specify your own settings for the ObjectCenter X resources for your particular machine by editing the **.Xdefaults** file in your home directory. After you edit your **.Xdefaults** file, you can reload your X resource database by issuing the following command:

```
xrdb -load ~/.Xdefaults
```

You can also modify the default settings for the ObjectCenter X resources for your particular machine by creating a file containing the new information and specifying that file with the **-config** switch when you invoke ObjectCenter.

You can override some of the resources by starting ObjectCenter with any of the command-line switches listed in Table 20 on page 227.

Syntax for resource definitions

In general, the format for the ObjectCenter resource definitions that you can modify is as follows:

*ResourceName***:** *ResourceValue*

where *ResourceName* can contain various combinations of client names, object names, and resource variables.

In general, *ResourceName* begins with the application name, which is **ObjectCenter**; in some cases, *ResourceName* begins with a component or object name.

This format corresponds to the most basic line that you can have in any X resource definition file. Note that a colon (**:**) and whitespace separate the name of the resource from the value of the resource.

Example

For instance, the following line specifies that all instances of the ObjectCenter Motif application have a background of **LightSteelBlue**:

```
ObjectCenter*motif*background:    LightSteelBlue
```

In this example, the *ResourceName* is **ObjectCenter*motif*background**, and *ResourceValue* is **LightSteelBlue.**

Similarly, the following specifies that the ObjectCenter object called ***window*frontscsreen** should have a background of **red**:

```
ObjectCenter*window*frontscreen.background:     red
```

To choose the Motif user interface style as your default for ObjectCenter, include the following line in your **.Xdefaults** file or your ObjectCenter application defaults file:

```
ObjectCenter*Model: Motif
```

Troubleshooting your .Xdefaults file

Before you run ObjectCenter, we recommend that you remove any global **.Xdefaults** file settings such as:

**ResourceName : Value*

These settings, such as:

```
*frameWidth : 1
*borderWidth: 1
```

set the *ResourceName* to *Value* for all instances of *ResourceName* in all programs that you run, including ObjectCenter. In this case, the X resource specifies the default border width and frame width as **1** for all applications.

Specifications like these in your .**Xdefaults** file will cause problems with ObjectCenter. Moreover, it is rarely appropriate to set X resources for all applications in this way.

Using mwm and "transient decorations"

Some window managers accept instructions not to decorate dialog boxes or other transient pop-ups. Decorating in this context means providing a border and a move bar; if a dialog box is not decorated it cannot be resized or moved.

For instance, if you are using the Motif window manager (**mwm**), the following entry in your .**Xdefaults** file instructs **mwm** not to decorate dialogs or other pop-up windows:

```
Mwm*transientDecoration:     none
```

We recommend that you remove this line from your **.Xdefaults** file. Otherwise, it will be difficult to use ObjectCenter's dialog boxes, since you will not be able to move them out of your way while keeping them on the screen.

NOTE See the “Bringing transient windows to the front” section on page 417 for additional information related to setting X resources.

Resource Descriptions

See Table 34 for a list of frequently used resources, along with a brief description of each, and their possible values.

Table 34 ObjectCenter X Resources and Their Possible Values

Name of Resource	Description	Possible Values
ObjectCenter*Color*OI_scroll_text .@text.Background	Sets color resources for scrolling text objects.	As specified by XLFD (X11 Logical Font Description)
ObjectCenter*Color*Workspace .@text.Background	Sets color resources for Workspace.	As specified by XLFD
ObjectCenter*ConfirmSelnUse	By default when an item on a menu, such as the **Examine** menu, is selected, the action is performed based on the current selection. Alternatively, the GUI can present a dialog box showing the current selection, which you can edit and then perform the action requested. If you want this alternative behavior, set the value of this resource to **True**.	**True** **False** (the default)
ObjectCenter*DefaultKey	Defines the accelerator key used to activate the default cell in a menu. When a cell is found with an accelerator that matches this key, the cell is made the default, and no accelerator label is displayed for it.	Any valid key

Table 34 ObjectCenter X Resources and Their Possible Values (Continued)

Name of Resource	Description	Possible Values
ObjectCenter*dimButtonsWhen DebuggerBusy	Specifies the length of time that the debugger must be busy for the control buttons on the GUI to dim. The default value is 1.15. The value of this resource must be: • The string Always if you want buttons to dim as soon as the debugger is busy • The string Never if you never want the buttons to dim. • Any positive floating-point number, to indicate the number of seconds you want to elapse before the buttons start to dim.	Always, Never, *number*
ObjectCenter*FixedWidthBoldFont	Changes the font of top-level entries.	As specified by XLFD
ObjectCenter*FocusPolicy	Specifies the focus policy to use. By default, the Motif GUI uses **click_to_type**, and OPEN LOOK uses **follows_pointer**. See the "Setting the keyboard input-focus" section on page 443 for more information.	**click_to_type** **follows_pointer**
ObjectCenter*Font	Changes fonts globally. See the "Examples of changing fonts in GUI components" section on page 461.	As specified by XLFD
ObjectCenter*HelpTranslations	Specifies the set of keys eligible for use as the help key. See the "Changing the help key" section on page 444 for more information.	Any valid sequence of keys and action functions

Table 34 ObjectCenter X Resources and Their Possible Values (Continued)

Name of Resource	Description	Possible Values
ObjectCenter*MainWindow*Examine MenuUsesPopups	Specifies whether or not the Print, Whatis, Whereis menu items on the Examine menu display their output in the Workspace (default) or in a special popup window.	**True** **False** (the default)
ObjectCenter*Model	Specifies the interaction model to use. The default setting for this resource is platform-specific. For instance, Hewlett-Packard workstations use **motif**, and Sun workstations use **openlook.**	**motif** **openlook** **openlook2d** **openlook3d**
ObjectCenter*MotifPushpin	Uses a "screw" to simulate OPEN LOOK pushpins. If you want a pushpin on dialogs in Motif, set this resource to **True**. See the "Adding pushpins to the Motif GUI" section on page 443 for more information.	**True** **False** (the default)
ObjectCenter*OI_entry_field.Font	Overrides the fonts for all single-line text entry fields.	As specified by XLFD
ObjectCenter*OI_multi_text.Font	Overrides the fonts for all multiple-line text entry fields.	As specified by XLFD
ObjectCenter*OI_scroll_text.Font	Overrides the fonts for all scrollable text objects.	As specified by XLFD
ObjectCenter*usePanner	The Inheritance Browser, Data Browser and Cross-Reference Browser use a scrollbar by default. To make them use a panner instead, set this resource to **True**.	**True** **False** (the default)

Table 34 ObjectCenter X Resources and Their Possible Values (Continued)

Name of Resource	Description	Possible Values
ObjectCenter*WMIgnoresPPosition	Notifies ObjectCenter that the window manager used does not properly interpret the PPosition bit in the **WM_NORMAL_HINTS** property. Set this resource if you are using **mwm** as a window manager.	**True** **False** (the default)
ObjectCenter*workspaceTranscriptSize	Specifies the maximum number of lines available for the Workspace. The default setting is **2000**. Setting the value to **0** means there is no maximum; that is, the number of lines in the Workspace can grow indefinitely.	Any integer greater than or equal to 0
ObjectCenter*XrefBrowser*showReturn Type	Specifies whether or not the Cross-Reference Browser shows the return type for functions.	**True** **False** (the default)

Examples of using resources

The examples in this section show you how to use X resources to do the following:

- Change the font size of text objects in the GUI
- Add pushpins to the Motif GUI
- Set the keyboard input-focus
- Change the help key

Changing the font size of text objects in the GUI

You can change the text fonts globally for all objects in the user interface with the following resources. In this example, the font size is set to 18 points (180 decipoints).

```
! Basic fonts for most OI objects

ObjectCenter*OI*font:-adobe-helvetica-medium-r-normal-*-*-180-*-*-*-*-*-*
ObjectCenter*OI*label.font:-adobe-helvetica-bold-r-normal-*-*-180-*-*-*-*
-*-*
ObjectCenter*OI*OI_scroll_menu.@title.font:-adobe-helvetica-bold-r-normal
-*-*-180-*-*-*-*-*-*
! Fonts for OI text objects: you should use a fixed-width font here
ObjectCenter*OI*OI_entry_field.font:-adobe-courier-medium-r-normal-*-*-18
0-*-*-*-*-*-*
ObjectCenter*OI*OI_multi_text.font:-adobe-courier-medium-r-normal-*-*-180
-*-*-*-*-*-*
ObjectCenter*OI*OI_scroll_text.font:-adobe-courier-medium-r-normal-*-*-18
0-*-*-*-*-*-*
! Special fonts for project and error browsers: use a fixed width font
here too
ObjectCenter*FixedWidthBoldFont:-adobe-courier-bold-r-normal-*-*-180-*-*-
*-*-*-*
```

Adding pushpins to the Motif GUI

The OPEN LOOK GUI offers you pushpins to keep menus and non-modal dialog boxes on the screen until you explicitly unpin them.

ObjectCenter's implementation of the Motif GUI provides *screws* as an equivalent for pushpins. By default they are not used; but if you run the Motif version of ObjectCenter with any window manager, you can enable the implementation of screws by including the following line in your .**Xdefaults** file:

```
ObjectCenter*MotifPushpin:    True
```

If you include this line, dialog boxes, but not menus, will have a screw in the lower right corner that you can select. When the dialog is "screwed in," it has the same semantics as if it were pinned; it is not removed when you press the Apply or OK buttons. It is removed when it is "unscrewed," or when the Cancel button is pressed.

Setting the keyboard input-focus

ObjectCenter's GUI provides two choices of keyboard input-focus: **click_to_type** and **follows_pointer**. By setting the GUI to **click_to_type**, you instruct the GUI to set keyboard focus when you click on an item; the focus will not move until you click on the next item. By setting the focus to **follows_pointer**, you instruct the GUI to set keyboard focus to the item under the mouse pointer. With either focus, you can still use explicit keyboard traversal commands to move the focus to pushbuttons, menus, and so on.

By default, the Motif GUI uses **click_to_type**, and OPEN LOOK uses **follows_pointer**.

If you want to specify your preference to be in effect in both the Motif and OPEN LOOK GUI, put one of the following lines in your X resources file:

```
ObjectCenter*focusPolicy:    follows_pointer
ObjectCenter*focusPolicy:    click_to_type
```

A disadvantage to setting the focus policy to **follows_pointer** is that you may find that, as you move the mouse pointer from one ObjectCenter window to another, the window under the mouse is sometimes automatically raised by the window manager (**mwm**) to the top; sometimes it is not. You can use the following resource setting to tell **mwm** not to raise windows when the mouse moves around the screen:

```
Mwm*focusAutoRaise:    false.
```

Changing the help key

By default, you request context-sensitive help with the Help or F1 key, depending on your platform. You can change the default setting with the HelpTranslations resource. For example, to use the F6 key as the Help key, put this line in your X resources file:

```
*OI*HelpTranslations: #override \n\\p <Key>F6: help()\n
```

Instead of **F6**, you could substitute any valid key sequence allowed in an X11 translation setting.

NOTE You cannot change the location of the scrollbar in ObjectCenter using X resources.

Run Window and Edit window resources

You can customize the Run Window, generic terminal window, or the **vi** editor window with any xterm resource. See the UNIX manual page for xterm for a complete list of xterm resources.

To specify a resource for the Run Window:

```
ObjectCenter*RunWindow.xterm-resource
```

For example, to save the content of the Run Window for your session to a file called **myfile.log** in your home directory, set the following resources:

```
ObjectCenter*RunWindow.logFile: myfile.log
ObjectCenter*RunWindow.logging: on
```

You can also turn logging on and off from the Run Window popup menu, which is displayed when you press the Left mouse button while holding down the Control key. If you don't specify a name for the log file, the session is saved in a file called **XtermLog.***xnnnn*.

To add a scrollbar to the Run Window with a 500-line scrolling history, set the following resource (the scrollbar resource is set to true by default). Note that some **xterm** resources should be set explicitly for the **vt100** subobject:

```
ObjectCenter*RunWindow*vt100.scrollBar: true
ObjectCenter*RunWindow*vt100.saveLines: 500
```

Set the following resources to change the default font and geometry of the Run Window:

```
ObjectCenter*RunWindow*vt100.font: <font specification>
ObjectCenter*RunWindow*vt100.geometry: <columns>x<rows>
```

If you invoke the **vi** editor from within ObjectCenter, you can customize it with any xterm resource. To do so:

```
ObjectCenter*EditWindow.xterm-resource
```

When the ObjectCenter user interface executes workspace commands, it uses a terminal to do so. You can customize this terminal with any xterm resource. To do so:

```
ObjectCenter*Terminal.xterm-resource
```

Setting window sizes

The general syntax for geometry resources is:

object_name.**geometry:** *W*x*H*[+*X*+*Y*]

where *W* and *H* are the width and height of the window (in either pixels or characters, depending on the application), and *X* and *Y* are the X and Y coordinates of the upper left-hand corner of the window, relative to the upper left-hand corner of the screen. You can also use -X and / or -Y instead of +X+Y, in which case you're setting the coordinates of the lower right-hand corner of the window, relative to the lower right-hand corner of the screen.

For instance, to make the Edit window 80 columns wide by 60 lines long, put the following line in your X resources file:

```
ObjectCenter*EditWindow.vt100.geometry: 80x60
```

Here are some more examples:

```
ObjectCenter*EditWindow.vt100.geometry: 80x40+20+30
```

This makes the Edit window 80 by 40 characters, 20 pixels from the left edge of the screen, and 30 pixels from the top.

```
ObjectCenter*RunWindow.vt100.geometry: 80x40+10-10
```

This makes the Run Window 80 by 40 characters, 10 pixels from the left edge, and 10 pixels from the bottom of the screen.

Keyboard editing

In addition to setting the resources already described, you may want to modify actions and translations in order to change the default settings for keyboard editing. Here we document our default settings and the translations needed to change the Motif defaults.

Default settings

By default, these shell-like and Emacs-like keybindings are available in ObjectCenter.

control-a	beginning of line
control-e	end of line
control-b	backward character
control-f	forward character
meta-b	backward word
meta-f	forward word
control-n	next line
control-p	previous line
control-d	delete next character
control-u	delete to beginning of line
control-k	delete to end of line
control-w	delete previous word

In Motif, some windows may use Meta-B and Meta-F as menu mnemonics, rendering them unavailable in text objects.

Changing Motif defaults for keyboard editing

If you are a Motif user and you want to change the default Motif settings for keyboard editing, you can use the actions and translations for the **OI_entry_field** objects shown inTable 35 and for **OI_multi_text** objects shown in Table 36.

For example, to add more emacs keyboard shortcuts, set the translations for the underlying objects in your X resources file:

```
ObjectCenter*OI_multi_text.Translations:  #override \n
 ~Shift ~Meta ~Ctrl <Key>Delete:          delete_previous_character()\n
 ~Shift ~Meta ~Ctrl <Key>BackSpace:       delete_previous_character()\n
               Meta <Key>D:               delete_next_word() \n
ObjectCenter*OI_entry_field.Translations: #override \n
 ~Shift ~Meta ~Ctrl <Key>Delete:          delete_previous_character()\n
 ~Shift ~Meta ~Ctrl <Key>BackSpace:       delete_previous_character()\n
               Meta <Key>D:               delete_next_word() \n
```

An **OI_entry_field** object is a region for entering or displaying a single line of text. Visually, it consists of an optional label followed by the text entry area. An **OI_multi_text** object is a viewport onto an underlying text structure consisting of zero or more lines. You can control the size of the viewport and manipulate the text displayed in the text object.

In Table 35 and Table 36, *selection* means characters that have been highlighted and are in the X window selection property, and **start-of-selection** means the character position in the **OI_entry_field** object in which the selection starts. **PRIMARY, SECONDARY** and **CLIPBOARD** selections are the selections stored in the X properties of the same names.

Table 35 OI_entry_field Translation Functions

Function Name	Description
backward_character( )	Moves the cursor left one character.
backward_view( )	Moves the cursor one viewport to the left (if the field is scrolled).
backward_word( )	Moves the cursor left one word. A word is delineated by whitespace.
beginning_of_line( )	Moves the cursor to the beginning of the entry.
cancel_select( )	If a selection is in progress using the mouse, deselects all the currently selected characters and terminates the selection process.
click_down( )	Processes button-down event in case click callback is set. Does not do any selection processing (see **select_start**).
click_up( )	Processes button-up event and dispatches to click callback if one is set. Does not do any text selection (see **select_end**).
copy_clipboard( )	Copies the currently selected text to the **CLIPBOARD** selection.
copy_primary( )	Inserts the **PRIMARY** selection at the current insertion point.
cut_clipboard( )	Copies the currently selected text to the **CLIPBOARD** selection and deletes the **PRIMARY** selection from its original source, if possible.
cut_primary( )	Inserts the **PRIMARY** selection at the current insertion point and deletes the **PRIMARY** selection from its original source, if possible.
delete_all_characters( )	Deletes all characters in the entry.
delete_next_character( )	Deletes the character to the right of the cursor.
delete_next_word( )	Deletes from the current insertion point through the whitespace at the end of the text containing the insertion point.

Table 35 OI_entry_field Translation Functions (Continued)

Function Name	Description
delete_previous_character( )	Deletes the character to the left of the cursor.
delete_previous_word( )	Deletes from the current insertion point up to, but not including, the whitespace at the beginning of the text containing the insertion point.
delete_to_beginning_of_line( )	Deletes all the characters to the left of the cursor.
delete_to_end_of_line( )	Deletes all the characters to the right of the cursor.
end_of_line( )	Moves the cursor to the end of the entry.
extend_start( )	Moves start-of-selection to the character under the mouse pointer.
focus_in( )	Sets input-focus to the object.
focus_out( )	Gives up input-focus.
forward_character( )	Moves the cursor right one character.
forward_view( )	Moves the cursor one viewport to the right (if the field can be scrolled).
forward_word( )	Moves the cursor right one word. A word is delineated by whitespace.
input_character( )	Inserts character at the cursor position.
insert_mode( )	Sets the character entry mode to **OI_EF_INSERT**.
insert_selection( )	Pastes text from the **PRIMARY** selection at the cursor position.
key_select( )	Marks the text from the anchor point to the cursor as the **PRIMARY** selection.
move_insertion( )	Sets the insertion point to the mouse pointer position; does not clear the current selection if it is in the same object.

Table 35 OI_entry_field Translation Functions (Continued)

Function Name	Description
move_selection( )	If the **SECONDARY** selection is active for this object, then copies the **SECONDARY** selection text to the cursor location and deletes the original selected text. Otherwise, if the **PRIMARY** selection is active for this object and was set using the mouse, copies the **PRIMARY** selection text to the cursor location and deletes the original selected text.
newline( )	End of entry. This causes the end-of-entry validation function to be called. If it returns **OI_EF_ENTRY_CHK_OK**, and an object has been registered via the **set_next** function, the new object obtains input-focus. Otherwise, the **OI_entry_field** object retains the input-focus.
next_object( )	Same as **newline( )**.
next_tab_group( )	Transfers the input-focus to the next tab group.
paste_clipboard( )	Deletes any currently selected text. The contents of the **CLIPBOARD** are then inserted in its place. If no text is currently selected, the contents of the **CLIPBOARD** are inserted at the insertion point.
previous_object( )	Completes the entry as if the Return key had been pressed and goes to the previous object, if one exists.
previous_tab_group( )	Transfers the input-focus to the previous tab group.
replace_mode( )	Sets the character entry mode to **OI_EF_REPLACE**.
replace_with_default( )	Replaces the current entry with the default entry.
scroll_left( )	Scrolls the text one full viewport to the left.
scroll_left_edge( )	Scrolls the text so the extreme left edge is visible.
scroll_right( )	Scrolls the text one full viewport to the right.
scroll_right_edge( )	Scrolls the text so the extreme right edge is visible.

Table 35 OI_entry_field Translation Functions (Continued)

Function Name	Description
secondary_adjust( )	Extends the selection to the new mouse pointer position. If Motif is being used, underlines the selection.
secondary_end( )	Completes the selection process and saves the selection in the **SECONDARY** selection.
secondary_start( )	Begins selecting text for inclusion in the **SECONDARY** selection.
select_adjust( )	Extends the selection to the mouse pointer location.
select_all( )	Marks the entire text as the **PRIMARY** selection.
select_end( )	Completes the selection process and saves the selection as the **PRIMARY** selection. Also, if the time between press and release is less than 500 milliseconds, calls click callback if one is registered.
select_start( )	Begins selecting text for inclusion in the **PRIMARY** selection at the mouse pointer location. Moves the cursor to the pointer position. Saves the time to determine if a button click occurred.
set_anchor( )	Sets the anchor for selection at the current insertion point.
take_focus( )	Sets the focus to the **OI_entry_field** object.
toggle_mode( )	Toggles the character entry mode between **OI_EF_INSERT** and **OI_EF_REPLACE**. The initial character entry mode is **OI_EF_INSERT**.
unselect_all( )	Deselects any currently selected text. Clears the **PRIMARY** selection if it is owned by the process.

Table 36 OI_multi_text Translation Functions

Function	Description
backward_character( )	Moves the cursor backwards one character. Wraps to the previous line if necessary. Repositions the text in the viewport if necessary.
backward_paragraph( )	Moves the cursor backwards one paragraph. Positions the cursor at the beginning of the paragraph. Repositions the text in the viewport if necessary.
backward_word( )	Moves the cursor backwards one word. Positions the cursor at the beginning of the word. Repositions the text in the viewport if necessary.
backward_view( )	Moves the cursor to the left edge of the viewport. If the cursor is already at the left edge of the viewport, shifts the text to display the next view to the left and positions the cursor at the left edge of the viewport.
beginning_of_file( )	Moves the cursor to the beginning of the text. Repositions the text in the viewport if necessary.
beginning_of_line( )	Moves the cursor to the beginning of the current line. Repositions the text in the viewport if necessary.
beginning_of_pane( )	Moves the cursor to the position in front of the first visible character in the first line currently visible in the viewport.
cancel_select( )	Cancels any selection that is in progress. Applies only while the mouse button is down.
click_down( )	Starts timing for a mouse button click.
click_up( )	Ends mouse button click timing and makes click callback.
copy_clipboard( )	Copies the **PRIMARY** selection (the currently selected text) to the **CLIPBOARD** selection.
copy_primary( )	Inserts contents of the **PRIMARY** selection at the cursor location.
cut_clipboard( )	Copies the **PRIMARY** selection to the **CLIPBOARD** selection, then deletes the **PRIMARY** selection.

Table 36 OI_multi_text Translation Functions (Continued)

Function	Description
cut_primary( )	Inserts the contents of the **PRIMARY** selection at the cursor location, then deletes the **PRIMARY** selection.
delete_all_characters( )	Deletes all the characters on the current line (the one the cursor is in), but leaves an empty line. Positions the cursor at the beginning of the line.
delete_next_character( )	If the cursor is at the end of the line, joins the following line with the current line; otherwise, deletes the character to the right of the cursor and closes up the space.
delete_next_word( )	If the cursor is at the end of the line, joins the following line with the current line; otherwise, deletes the word to the right of the cursor and closes up the space.
delete_previous_character( )	If the cursor is at the beginning of the line, joins the current line with the previous line; otherwise, deletes the character to the left of the cursor and closes up the space.
delete_previous_word( )	If the cursor is at the beginning of the line, joins the current line with the previous line; otherwise, deletes the word to the left of the cursor and closes up the space.
delete_this_line( )	Deletes the line the cursor is in and closes up the space.
delete_to_beginning_of_line( )	Deletes from the beginning of the current line to the cursor position and closes up the space.
delete_to_end_of_line( )	Deletes from the cursor position to the end of the current line and closes up the space.
end_of_file( )	Moves the cursor to the end of the text. Repositions the text in the viewport if necessary.
end_of_line( )	Moves the cursor to the end of the current line. Repositions the text in the viewport if necessary.
end_of_pane( )	Moves the cursor to the position after the last character on the last line currently visible.

Table 36 OI_multi_text Translation Functions (Continued)

Function	Description
extend_start( )	Begins a **PRIMARY** selection as for **select_start**, but initially includes all text from the mouse pointer to the cursor.
focus_in( )	Paints focus indicators to indicate that the object has the input-focus.
focus_out( )	Paints focus indicators to indicate that the object does not have the input-focus.
forward_character( )	Moves the cursor forward one character. Wraps to the next line if necessary. Repositions the text in the viewport if necessary.
forward_paragraph( )	Moves the cursor forward one paragraph. Positions the cursor at the beginning of the paragraph. Repositions the text in the viewport if necessary.
forward_word( )	Moves the cursor forward one word. Positions the cursor at the beginning of the word. Repositions the text in the viewport if necessary.
forward_view( )	Moves the cursor to the right edge of the viewport. If the cursor is already at the right edge of the viewport, shifts the text to display the next view to the right and position the cursor at the right edge of the viewport.
input_character( )	Inserts the character at the current cursor position. Works only for key events.
input_convert( )	Works only for key events. Passes the key event to the language server for conversion. When the server is done converting (this may take several key events), inserts the result at the cursor position.
insert_mode( )	Puts the object in insert mode. This means that any characters inserted are placed at the current cursor location and are inserted between the two adjacent characters.

Table 36 OI_multi_text Translation Functions (Continued)

Function	Description
insert_selection(*arg***)**	Inserts text from an X selection at the cursor position. If no arguments are present, the **PRIMARY** selection is used. Otherwise, *arg* is used as the name of the selection to use.
insert_string(*arg***)**	Inserts the string *arg* at the cursor location. Use "\n" to indicate newlines and the Tab key (not "\t") to indicate tabs when specifying the string *arg*.
key_select( )	Puts all the text between the point marked by **set_anchor** to the current cursor location in the **PRIMARY** selection.
move_insertion( )	Moves the cursor to the location under the mouse pointer without clearing the **PRIMARY** selection.
move_selection( )	If the **SECONDARY** selection is active for this object, then copies the **SECONDARY** selection text to the cursor location and deletes the original selected text. Otherwise, if the **PRIMARY** selection is active for this object and was set using the mouse, copies the **PRIMARY** selection text to the cursor location and deletes the original selected text.
newline( )	If an end-of-entry callback is registered, calls it. If no end-of-entry callback exists or if the end-of-entry callback returned **OI_mt_entry_chk_ok**, then inserts a new line after the line the cursor is in, and positions the cursor at the beginning of the new line.
next_line( )	Moves the cursor to the next line. Repositions the text in the viewport if necessary.
next_object( )	Sets the input-focus to the next object in the focus chain (if any).
next_tab_group( )	Sets the input-focus to the focus object in the next tab group (if any).
next_page( )	Scrolls the text so that the next page towards the end of the text becomes visible.

Table 36 OI_multi_text Translation Functions (Continued)

Function	Description
open_next_line( )	Inserts a blank line following the line containing the cursor. Rearranges the text accordingly.
open_previous_line( )	Inserts a blank line previous to the line containing the cursor. Rearranges the text accordingly.
paste_clipboard( )	Inserts the text from the **CLIPBOARD** selection at the cursor location.
previous_line( )	Moves the cursor to the previous line. Repositions the text in the viewport if necessary.
previous_object( )	Sets the input-focus to the previous object in the focus chain.
previous_tab_group( )	Sets the input-focus to the focus object in the previous tab group, if any.
previous_page( )	Scrolls the text so that the previous page towards the beginning of the text becomes visible.
process_return( )	If an end-of-entry callback is registered, calls it. If no end-of-entry callback exists or if the end-of-entry callback returned **OI_mt_entry_chk_ok**, then positions the cursor at the beginning of the next line. If there is no next line (that is, the cursor is in the last line of text), inserts a new line at the end of the text and positions the cursor there.
replace_mode( )	Puts the object in replace mode. This means that any characters input replace the character to the right of the current cursor location. Any characters input at the end of the line are appended to the line.
scroll_bottom( )	Scrolls to the last line of the text.
scroll_down( )	Scrolls one full viewport towards the end of the text.
scroll_left( )	Scrolls one full viewport towards the left of the text.
scroll_left_edge( )	Scrolls to the first character in the line.
scroll_right( )	Scrolls one full viewport towards the right of the text.

Table 36 OI_multi_text Translation Functions (Continued)

Function	Description
scroll_right_edge( )	Scrolls to the last character position in the object. This may be well past the last character in the current line.
scroll_top( )	Scrolls to the first line of the text.
scroll_up( )	Scrolls one full viewport towards the beginning of the text.
secondary_adjust( )	Adjusts the **SECONDARY** selection to include all text from the **secondary_start** position to the position under the mouse pointer.
secondary_end( )	Completes the selection process and saves the selection as the **SECONDARY** selection. Also processes button-up event and calls click callback if one is registered.
secondary_start( )	Begins selecting text at the current location under the pointer for the **SECONDARY** selection. Underlines the selected text.
select_adjust( )	Adjusts the **PRIMARY** selection to include all text from the **select_start** position to the position under the pointer.
select_all( )	Puts the entire text in the **PRIMARY** selection. Highlights the selection.
select_end( )	Completes the selection process and saves the selection as the **PRIMARY** selection. Also processes button-up event and calls click callback if one is registered.
select_line( )	Selects the entire current line as the **PRIMARY** selection.
select_start( )	Begins selecting text at the mouse pointer location for the **PRIMARY** selection. Highlights the selection. Also processes button-down event in case click callbacks are registered.
set_anchor( )	Marks the current cursor location as the start for a keyboard-defined selection.

Table 36 OI_multi_text Translation Functions (Continued)

Function	Description
start_input_conversion( )	Sends all subsequent characters to the input server for conversion.
stop_input_conversion( )	Treats subsequent characters normally (quits sending characters to the input server).
toggle_mode( )	If the current mode is insert mode, changes it to replace mode. If the current mode is replace mode, changes it to insert mode.
take_focus( )	Sets the focus to the **OI_multi_text** object.
unselect_all( )	Unselects any currently selected text.

Using component and object names

When you specify a resource name, you can use object names as well as application names. Furthermore, since ObjectCenter is built using the OI (Object Interface) toolkit, its object names include some OI names as well as names of components specific to ObjectCenter. See Table 37 for a list of object and component names that you can use when you specify ObjectCenter resources.

Table 37 Component and Object Names Used to Set X Resources

General Area of User Interface	Specific Component of Interface	Name of Resource
Main Window	Primary Window	**topApp**
	Source Panel	**sourcePanel**
	Workspace	**workspacePanel**
	Breakpoint Icon	**BreakGlyph**
	Action Icon	**ActionGlyph**
	Breakpoint Location Arrow	**ArrowGlyph**
	Scope Location Arrow	**HArrowGlyph**
Data Browser	Primary Window	**DataBrowser**
	Background Canvas	**layoutMgr**
	Data Items	**dataCell**
	Resize Corners	**ResizeLL** (lower left) **ResizeLR** (lower right)
	Pointer Follow Boxes	**ReferenceGlyph**
Cross-Reference Browser	Primary Window	**XrefBrowser**
	Background Canvas	**layoutMgr**
	Xref Nodes	**SCrossCell**
	Follow Boxes	**ReferenceGlyph**

Table 37 Component and Object Names Used to Set X Resources (Continued)

General Area of User Interface	Specific Component of Interface	Name of Resource
Error Browser	Primary Window	**errBrowser**
Options Browser	Primary Window	**optBrowser**
Project Browser	Primary Window	**projectWindow**
Thread Browser	Primary Window	**threadBrowser**
Inheritance Browser	Primary Window Hierarchy Browser	**hierarchyBrowser**
	Primary Window Class Examiner	**classExaminer**
	Background Canvas	**layoutMgr**
OI objects	Entry Field	**OI_entry_field**
	Entry Field Label	**OI_entry_field.label**
	Multi Text	**OI_multi_text**
	Scroll Text	**OI_scroll_text**
	Sequenced Entry Field	**OI_seq_entry_field**
	Seq Entry Field Label	**OI_seq_entry_field.label**
	Abbreviated Menu	**OI_abbr_menu**
	Button Menu	**OI_button_menu**
	Menu Cell	**OI_menu_cell**
	Exclusive Check Menu	**OI_excl_check_menu**
	Exclusive Rect Menu	**OI_excl_rect_menu**
	Poly Check Menu	**OI_poly_check_menu**
	Poly Rect Menu	**OI_poly_rect_menu**
	Scroll Menu	**OI_scroll_menu**
	Static Text	**OI_static_text**

Examples of changing fonts in GUI components

Here are specific examples for changing the fonts of the text in some important components of the GUI:

```
ObjectCenter*DataBrowser*layoutMgr*font: 5x8
ObjectCenter*XrefBrowser*layoutMgr*font: 5x8
ObjectCenter*workspacePanel*font: 5x8
ObjectCenter*sourcePanel*font: 5x8
```

Examples of using OI components

In addition to the general names listed in Table 37, OI adds some additional names you can use to specify the scope of a resource setting. These names are part of the *OI resource stack*, a list of prefixes that tells ObjectCenter to which objects, classes, or applications it should apply a resource. This list has a default hierarchy of elements, which is shown in Table 38. They are listed from most general to least general, from top to bottom.

Table 38 Elements of the OI Resource Stack Used to Specify X Resources

Instance	Class
Application name	Application class
oi	**OI**
color or **monochrome**	**Color** or **Monochrome**
screen number (example: **screen0**)	Screen number (example: **Screen0**)
language (example: **defaultLanguage**)	Language (example: **DefaultLanguage**)
openlook2d, **openlook3d**, or **motif**	**Openlook** or **Motif**
object hierarchies (instance names)	Object hierarchies (class names)

The most useful of the OI resources are probably the following:

- **color** vs. **monochrome**
- **openlook** vs. **openlook3d** vs. **motif**

For instance, you can set a resource only on a color machine as follows:

```
ObjectCenter*color*Background:  red
```

Similarly, if you want to specify the background resource only on a color machine in Motif, you could say:

```
ObjectCenter*color*motif*Background:  red
```

Defining GUI-specific resources

If you want to define ObjectCenter resources differently for OPEN LOOK vs. Motif, you can specify them uniquely by using the resources shown in Table 38. The following syntax sets the *Resource* to *Value* for the Motif model:

ObjectCenter*motif*Resource* : *Value*

The following syntax sets the *Resource* to *Value* for the OPEN LOOK 2D model. (2D is typically used on monochrome machines.)

ObjectCenter*openlook2d*Resource* : *Value*

The following syntax sets the *Resource* to *Value* for the OPEN LOOK 3D model. (3D is typically used on color machines.)

ObjectCenter*openlook3d*Resource* : *Value*

For instance, to set all entry fields' fonts to 8x10 for the Motif model, but make them 12x14 for OPEN LOOK3D, specify these two resources:

```
ObjectCenter*motif*OI_entry_field.font:         8x10
ObjectCenter*openlook3d*OI_entry_field.font:  12x14
```

To set the entry fields' background to red for all models of ObjectCenter:

```
ObjectCenter*OI_entry_field.background:    red
```

Setting resources for scrolling text objects

You can set resources for scrolling text objects such as the Source area. For example, you can set the background color of all scrolling text objects to yellow and the foreground to blue with this syntax:

```
ObjectCenter*Color*OI_scroll_text.@text.Background:Yellow
ObjectCenter*Color*OI_scroll_text.@text.Foreground:Blue
```

You can also specify colors in the Workspace with this resource:

```
ObjectCenter*Color*Workspace.@text.Background: Azure
ObjectCenter*Color*Workspace.@text.Foreground: Ivory
```

User-defined commands

ObjectCenter provides two ways to define your own commands:

- Using the User Defined dialog box from the ObjectCenter menu on the Main Window.
- Using X11 resources to add commands to the User Defined menu in the Project Browser.

NOTE The commands that you define using X resources are different from the commands that you define using the User Defined dialog box. In general, we recommend that you use the User Defined dialog box to create commands, rather than using X resources. Commands created using the GUI are stored in your **.octruscmd** file in your home directory. See the *ObjectCenter User's Guide* for information about the **User Defined Commands** menu.

In this section we describe how to use X11 resources to define your own commands.

The ObjectCenter Project Browser allows you to specify 20 user-defined commands using X11 resources. Their internal names are **UserCmd1** through **UserCmd20**. ObjectCenter uses the internal name to look up the resources that describe each command. You can name the commands whatever you want to appear on the pulldown menu.

Examples of user-defined commands

To specify a user-defined command, you generally have to write only two lines per command, one for the label and another for the command.

For instance, the following two lines in an ObjectCenter **app-defaults** file will create a user-defined button labeled "Find Locked Files" that runs **listlocks**:

```
*ProjectBrowser.UserCmd1.label: Find Locked Files
*ProjectBrowser.UserCmd1.command: listlocks
```

If you do not have an ObjectCenter **app-defaults** file and you want to put this in your **.Xdefaults** file instead, start each line with "ObjectCenter". For instance:

```
ObjectCenter*ProjectBrowser.UserCmd1...
```

The next two lines will create a second button labeled "List Files" that will run **ls -lg** on all of the selected files:

```
*ProjectBrowser.UserCmd2.label: List Files
*ProjectBrowser.UserCmd2.command: ls -lg $files
```

And here is one that will run **ls -lg** on all of the selected sources instead:

```
*ProjectBrowser.UserCmd3.label: List Sources
*ProjectBrowser.UserCmd3.command: ls -lg $sources
```

The next example runs **emacs** on all of the selected sources. Note that you can use the **useTerminalEmulator** resource to avoid running an extra **xterm**:

```
*ProjectBrowser.UserCmd4.label: Edit Sources
*ProjectBrowser.UserCmd4.command: emacs $sources
*ProjectBrowser.UserCmd4.useTerminalEmulator: False
```

The following example runs **listlocks** to find all the locked files, and waits until it is done before allowing the Project Browser to go on. The example also puts a menu separator bar just before this item.

```
*ProjectBrowser.UserCmd5.label: List Locked Files
*ProjectBrowser.UserCmd5.command: listlocks
*ProjectBrowser.UserCmd5.waitUntilDone: True
*ProjectBrowser.UserCmd5.addSeparator: True
```

X resources for user-defined commands

See Table 39 for a description of the resources to use when defining your own commands.

Table 39 X Resources for User-Defined Commands

Resource Class	Resource Name	Type
Label	label	String
This is the string shown in the pulldown menu item for this command. You should specify this item for each command, but if you do not, it defaults to the value of the command line itself.		
Command	command	String
This is the command line you want to execute when the menu item is selected. The text you supply can include any of the special words listed in Table 40 on page 467.		
WaitUntilDone	waitUntilDone	Boolean
A value of **True** means the Project Browser should wait until the command has finished before continuing. A value of **False** means the Project Browser will spawn the specified command and then continue immediately. The default value is **False**, so you only need to specify this resource if you want the Project Browser to wait.		
UseTerminalEmulator	useTerminalEmulator	Boolean
A value of **True** means the command must be run in a terminal emulator window, while a value of **False** means the command either does not require a window, or runs in its own window. The default value is **True**, so you only need to specify this item if you want to run a program like **emacs** that opens its own window.		
AddSeparator	addSeparator	Boolean
A value of **True** means insert a menu separator immediately before this item in the pulldown menu. A value of **False** means do not insert a separator. This is provided so you can build menus that are divided into categories of related commands, with separators between them. The default value is **False**, so you only need to specify this item if you want a separator.		

Table 39 X Resources for User-Defined Commands (Continued)

Resource Class	Resource Name	Type
TerminalEmulator	terminalEmulator	String

This string identifies the terminal emulator to use when running the program. The terminal emulator specification may include the following special words:

$program	Replaced with the pathname of a shell script that is to be executed. This script contains the full text of the expanded command line.
$command	Replaced with the text of the command label as shown in the menu item. This might be used to set the window title of the terminal emulator, for example.

If the $**program** string is not found in the terminal emulator specification, then the Project Browser automatically appends the name of the shell script to the end of the string. If you do not specify a value for **TerminalEmulator**, a suitable default value will be used, so you only need to specify this item if you want a custom terminal emulator applied to a particular command. The default value for the terminal emulator is:

clxterm -T $command -n $command -sb -sl 1000 -e

You can change the default value by setting the following resource:

```
ObjectCenter*ProjectBrowser.DefaultTerminalEmulator
```

See Table 40 for a list of the special words you can use in a **command** Resource.

Table 40 Special Words Used in the **command** Resource

Area of Interface	Special Word	What Replaces the Special Word
Project Browser	**$pwd**	ObjectCenter's current working directory.
	$files	A space-separated list of the full pathnames for all the files that are currently selected in the Project Browser.
	$**sources**	Like $**files**, except that the names of object files are replaced with the full pathnames of the corresponding source files, if the names are known. If an object file is selected for which the source file is unknown, nothing is generated for that file.
	$**libraries**	Replaced with a space-separated list of the full pathnames for all the libraries that are selected in the Project Browser.
	$**command**	Replaced with the text of the command label as shown in the menu item. When you execute a user-defined command, the command string is expanded into a shell script, which is then run using **execvp**, so your **PATH** and environment variable settings are available from within user-defined commands.
Main Window	**$filename**	The filename of the file in the Source area, relative to ObjectCenter's current working directory.
	$filepath	The absolute filename of the file in the Source area.
	$first_selected_char	The position of the first character selected on **$first_selected_line**. Character positions are numbered beginning with 1, and tabs are considered to be a single character. If no text is selected in the Source area, this keyword returns 0.

Table 40 Special Words Used in the **command** Resource (Continued)

Area of Interface	Special Word	What Replaces the Special Word
	$first_selected_line	Starting line number of the Source area's current text selection. Lines are numbered beginning with 1. If no text is selected in the Source area, this keyword returns 0.
	$last_selected_char	The position of the last character selected on **$last_selected_line**. Character positions are numbered beginning with 1, and tabs are considered to be a single character. If no text is selected in the Source area, this keyword returns 0.
	$last_selected_line	Ending line number of the Source area's current text selection. Lines are numbered beginning with 1. If no text is selected in the Source area, this keyword returns 0.
Current selection	**$selection**	Replaced with the current contents of the X11 **PRIMARY** selection, interpreted as a string. If the current selection is not available or is empty, **$selection** is replaced with an empty string.
Clipboard	**$clipboard**	Replaced with the current contents of the X11 **CLIPBOARD** selection. If the current selection is not available or is empty, **$clipboard** is replaced with an empty string.

Revision control systems

To support simple revision control systems, ObjectCenter defines four additional user-defined commands that have standard names. The internal names for these commands are as follows: **CheckIn**, **CheckOut**, **FileHistory**, and **FileDiffs**.

They are exactly like the user-defined commands described in the preceding section, and they read all of the same resources, but they have standard values for the **label** resource so that in general you only have to specify the command-line text.

For instance, to specify that you want your **checkin** command to be as follows:

```
ci -l
```

meaning RCS check in, then check out locked, you can say:

```
*ProjectBrowser.CheckIn.command: ci -l $sources
```

As a result, a button labeled “Check Files In” will appear. When you select this button, ObjectCenter issues the **ci -l** *files* command.

The standard labels are “Check Files In”, “Check Files Out”, “Show File Histories”, and “Show File Diffs”. You can change them by using the standard **label** resource. For instance, the following line:

```
*ProjectBrowser.CheckIn.label: Check In And Reacquire
```

changes the label of the **CheckIn** button.

Higher-level support for rcs, sccs, and reserve/replace

For convenience, to eliminate the need to specify four command settings for the four standard buttons, there is a single resource you can set that tells the Project Browser to use standard values for the **rcs** and **sccs** commands.

The resource is named ***ProjectBrowser.RevisionControl**, its type is **String**, and its default value is **sccs**. You can change the default to **rcs** or **None**.

See Table 41 for a list of the values for the four standard revision control commands according to the value of this resource.

Table 41 Values for Revision Control Commands Using ***ProjectBrowser.RevisionControl**

Revision Control Command	Value if Resource Set to rcs	Value if Resource Set to sccs
checkout	**co -l $sources**	**sccs edit $sources**
checkin	**ci -u $sources**	**sccs delget $sources**
history	**rlog $sources**	**sccs prs $sources**
diff	**rcsdiff -c $sources**	**sccs diffs -C $sources**

For example, you can edit your **.Xdefaults** file, adding one line that says:

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

to enable the user-defined commands for **rcs**; they will appear on the **User Defined** menu.

NOTE If you want to set up your own revision control commands, set the default value of the **ProjectBrowser.RevisionControl** resource to **None.** If you do so, and you have no other commands defined, the **User Defined** menu is removed.Also, the Project Browser support for **sccs** assumes you have a **sccs** wrapper program. If you don't have one, you may be able to get an **sccs** wrapper program free from BSD sources off the network (**uunet.uu.net**).

X Windows and customization for DynaText

The DynaText online documentation browser is designed to run with any ICCCM-compliant window manager and with any X server. Thus, it should display on most X terminals or on OpenWindows.

Font locations

The X Windows system in general looks for its fonts in specific locations and in locations you can specify. See your X documentation for details. The DynaText system, built on top of X, will look in the same places. In addition, DynaText uses a set of its own fonts. These fonts are located in the following directory:

*/path_to/***CenterLine/doc/data/X***platform***/fonts**

The word *platform* designates your platform' s particular implementation of X, such as 11 for X11.

DynaText ships only a special font for the Table of Contents view (the "annotation" font for characters like checkmark) and fonts for equation rendering. If for some reason the annotation font cannot be found, the Table of Contents view will simply use other symbols.

To make sure your X terminal or display station has access to the DynaText fonts, you can mount the fonts directory using the UNIX mount command. If you cannot mount this remote directory on your local machine, you must install the necessary fonts. Use whatever technique is available on your display station to install fonts. The BDF forms, the basic X ASCII font format, can be found in

*/path_to/***CenterLine/doc/data/X11/fonts**

Most X servers will either read these fonts directly, or provide a converter from BDF to their proprietary font format.

Font selection

Font specification in DynaText deviates a bit from the X paradigm. Although font specifications in Xdefaults are handled in the standard manner, font specifications in stylesheets deviate from the standard X mechanisms. DynaText allows for the specification of fonts by their components: family, weight, slant, and size.

One tool you can use when choosing fonts is called **xfontsel**, provided on the MIT core distribution tape. This program allows you to select and view fonts by their components. For example, you can see all of the available Helvetica-Bold sizes, and view each one individually.

Font sizes

Font sizes under X are a function of several factors: your machine's screen resolution, the order of font directory names in your font search path, whether the font exists at the size specified, whether you are running X11R5 or not, and whether you specify outline fonts or not.

Screen resolution

Prior to X11R5, all X fonts were raster fonts that were created at each point size for each screen resolution. For example, there is commonly a 10-point times font for a 75-dpi screen and a 10-point font for a 100-dpi screen. X uses the closest approximation for screens of different resolutions, such as 85-dpi.

Font path

Even if you have the 100-dpi fonts at all desired point sizes, if the 75-dpi fonts precede the 100-dpi fonts in your font path, X will use the 75-dpi fonts. For resetting your font path to the desired order, see the X manual page for the **xset** command.

Nonexistent point sizes

If the given font does not exist at the specified point size, X falls back on its default font, usually courier. X requires that you use an existing point size in the font.

Proportional fonts

At X11R5, you can use outline fonts, if they exist on your system and on your target systems. Using outline fonts solves the problem of nonexistent sizes, as the X server will construct the font at the appropriate size. You must, however, check that using nonstandard sizes does not adversely impact rendering performance. X11R5 also allows you to dedicate one machine on your network as a font server. See the X11R5 documentation for the **fontserver** command.

X application defaults

All X application defaults for the DynaText system are stored in the **defaults** subdirectory of the following directory:

*/path_to/***CenterLine/doc/data/X***platform*

For example, if you are running the Motif version, your X application defaults are stored in this directory:

*/path_to/***CenterLine/doc/data/X11/defaults**

Table 42 lists the appropriate settings for your system's user interface.

Table 42 Settings for X Implementations

X implementation	Setting
Motif user interface with MIT fonts	**X11**
Motif user interface with OpenWindows fonts (This setting used to be called Xweird.)	**Xmol**
Motif user interface with HP fonts	**Xhp**
Motif user interface with IBM fonts	**Xibm**

One group of files in the following directory concerns the DynaText Browser application:

*/path_to/***CenterLine/doc/data/X***platform***/defaults/C**

The file **Dtext.color** in this directory contains the color defaults; **Dtext.mono** contains the monochrome defaults. **Dtext** is a symbolic link to the one currently in effect.

Feel free to change any X settings you wish. Remember, an X application must apply settings in two steps:

1. Apply the application defaults from

   */path_to/***CenterLine/doc/data/X***platform***/defaults/C**

2. Apply any defaults from your own **.Xdefaults** file.

Thus, you can modify defaults by changing the system defaults or your **.Xdefaults**. See Table 43 for more information on the particular settings available.

Colors

Often when working with DynaText you will need to select colors. This might occur when customizing your X defaults.

If you are running the Motif version, the colors available to you are stored in this file:

/usr/lib/X11/rgb.txt

In addition, you can specify colors using a triplet of rgb (red, green, and blue) values, preceded by a pound sign. For example **#aabb00** specifies the color whose red value is hex **aa**, blue value is hex **bb**, and green value is hex **00**.

Any color specification in DynaText can be specified in either of these two ways: using a string from /**usr**/**lib**/**X11**/**rgb.txt**, or using an rgb triplet.

X resource names

Table 43 contains examples of DynaText settings. These examples introduce you to some of the named X objects for which you can change resources. If you wish, you can experiment with properties other than geometry. See the following file for documentation of other properties:

*/path_to/***CenterLine/doc/data/X***platform***/defaults/C/Dtext**

Consult your X Window System documentation for full details on the syntax of properties and their values.

Table 43 DynaText Settings and Descriptions

Setting	Description
***dynatext.geometry:**	geometry of the main library window
***dynatext*foreground:**	foreground of the main library window
***dynatext*background:**	background of the main library window
***dialog*foreground:**	foreground of dialog boxes
***dialog*background:**	background of dialog boxes
***menu_pane*foreground:**	foreground of popup menus
***menu_pane*background:**	background of popup menus
***note.geometry:**	geometry of annotation note
***note*foreground:**	foreground of annotation notes
***note*background:**	background of annotation notes
***log.geometry:**	geometry of message log
***log*background:**	background of message log

Table 43 DynaText Settings and Descriptions (Continued)

Setting	Description
***log*foreground:**	foreground of message log
***annotwin.geometry:**	geometry of annotation manager
***annotwin*foreground:**	foreground of annotation manager
***annotwin*background:**	background of annotation manager
***history.geometry:**	geometry of history window
***history*foreground:**	foreground of history window
***history*background:**	background of history window
***raster.geometry:**	geometry of raster images
***vector.geometry:**	geometry of vector images
***ftwin_main*toc_scrollwin.height:**	height of Table of Contents pane in fulltext window
***ftwin_main*toc_scrollwin.width:**	width of Table of Contents pane in fulltext window
***name*_v.geometry:**	geometry of fulltext view with name *name* when orientation is top-bottom
***name*_h*foreground:**	foreground of fulltext view with name *name* when orientation is left-right
***name*_v*background:**	background of fulltext view with name *name* when orientation is top-bottom
***name*_h*background:**	background of fulltext view with name *name* when orientation is left-right

Appendix A GNU General Public License

This appendix contains the GNU General Public License, which applies to the CenterLine GNU Debugger (pdm) and the CenterLine C preprocessor (clpp).

GNU General Public License

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1 This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2 You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3 You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4 You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5 You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6 You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These

actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7 Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8 If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author / donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9 If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10 The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11 If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12 BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU

ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13 IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Applying These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

ONE LINE TO GIVE THE PROGRAM'S NAME AND AN IDEA OF WHAT IT DOES
Copyright (C) 19YY NAME OF AUTHOR

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

SIGNATURE OF TY COON, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

This index covers the ObjectCenter User's Guide (page numbers prefaced with U) and the ObjectCenter Reference (page numbers prefaced with R).

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z