Hands On: C/C++ Programming and Unix Application Design: UNIX System Calls and Subroutines using C, Motif, C++

Contents

Chapter 1

The Common Desktop Environment

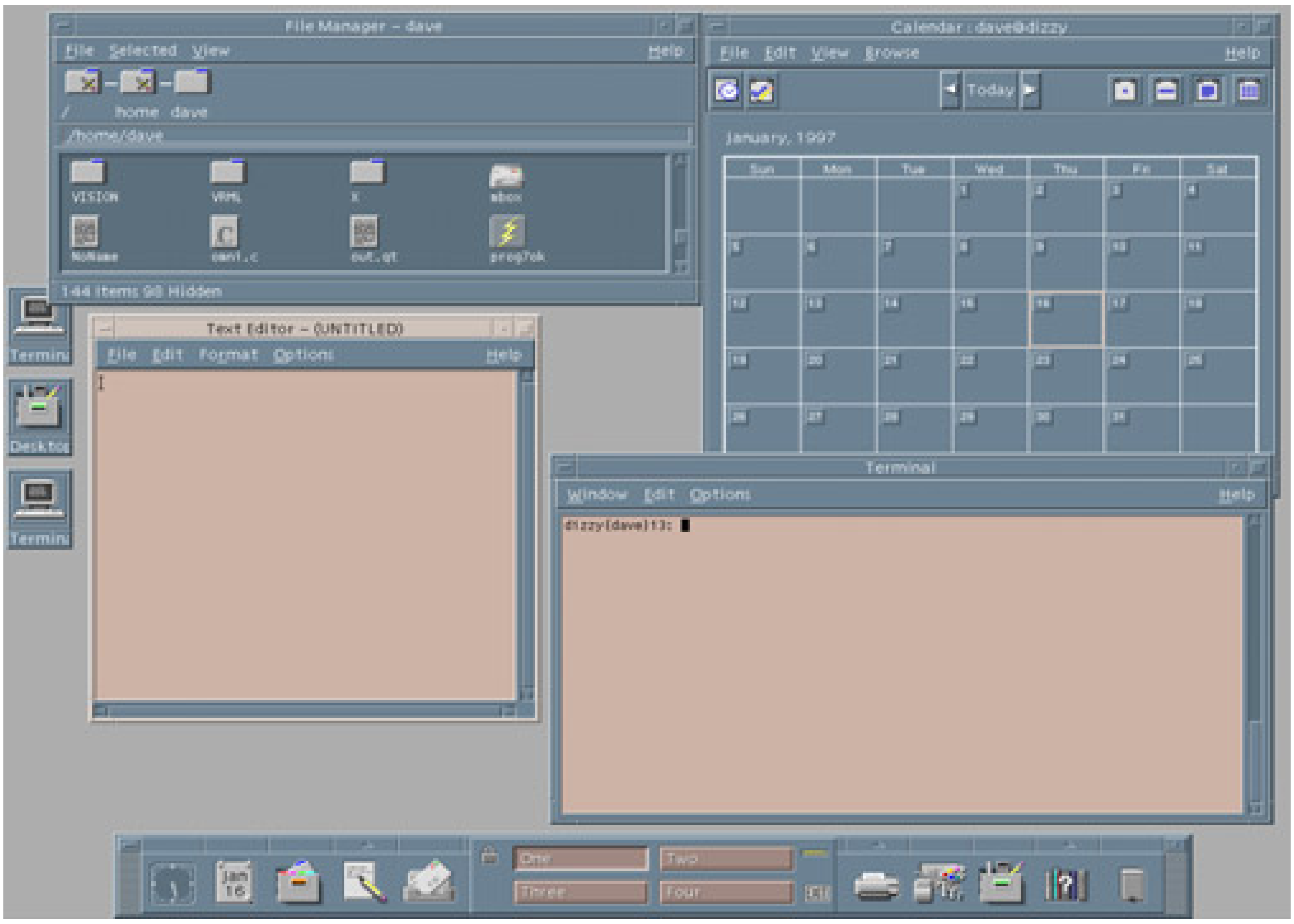
In order to use Solaris and most other Unix Systems you will need to be familiar with the Common Desktop Environment (CDE). Before embarking on learning C with briefly introduce the main features of the CDE.

Most major Unix vendors now provide the CDE as standard. Consequently, most users of the X Window system will now be exposed to the CDE. Indeed, continuing trends in the development of Motif and CDE will probably lead to a convergence of these technologies in the near future. This section highlights the key features of the CDE from a Users perspective.

Upon login, the user is presented with the CDE Desktop (Fig. 1.1). The desktop includes a front panel (Fig. 1.2), multiple virtual workspaces, and window management. CDE supports the running of applications from a file manager, from an application manager and from the front panel. Each of the subcomponents of the desktop are described below.

1.1 The front panel

The front panel (Fig. 1.2) contains a set of icons and popup menus (more like roll-up menus) that appear at the bottom of the screen, by default (Fig. 1.1). The front panel contains the most regularly used applications and tools for managing the workspace. Users can drag-and-drop application icons from the file manager or application manager to the popups for addition of the application(s) to the associated menu. The user can also manipulate the



```
dizzy{dave}13:
```

Figure 1.1: Sample CDE Desktop

default actions and icons for the popups. The front panel can be locked so that users can't change it. A user can configure several virtual workspaces — each with different backgrounds and colors if desired. Each workspace can have any number of applications running in it. An application can be set to appear in one, more than one, or all workspaces simultaneously.

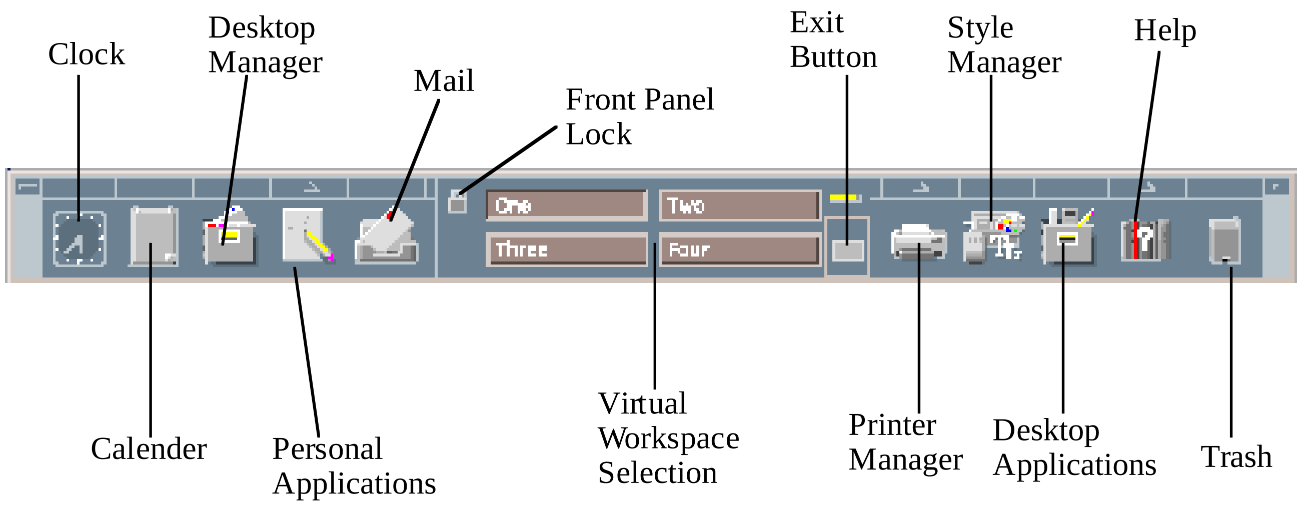


Figure 1.2: The CDE Front Panel

1.2 The file manager

CDE includes a standard file manager. The functionality is similar to that of the Microsoft Windows, Macintosh, or Sun Open Look file manager. Users can directly manipulate icons associated with UNIX files, drag-and-drop them, and launch associated applications.

1.3 The application manager

The user interaction with the application manager is similar to the file manager except that is is intended to be a list of executable modules available to a particular user. The user launches the application manager from an icon in the front panel. Users are notified when a new application is available on a server by additions (or deletions) to the list of icons in the application manager window. Programs and icons can be installed and pushed out to other workstations as an integral part of the installation process. The list of workstations that new software is installed on is configurable. The application manager comes preconfigured to include several utilities and programs.

1.4 The session manager

The session manager is responsible for the start up and shut down of a user session. In the CDE, applications that are made *CDE aware* are warned via an X Event when the X session is closing down. The application responds by returning a string that can be used by the session manager at the user's next login to restart the application. CDE can remember two sessions per user. One is the *current* session, where a snapshot of the currently running applications is saved. These applications can be automatically restarted at the user's next login. The other is the default login, which is analogous to starting an X session in the Motif window manager. The user can choose which of the two sessions to use at the next login.

1.5 Other CDE desktop tools

CDE 1.0 includes a set of applications that enable users to become productive immediately. Many of these are available directly from the front panel, others from the desktop or personal application managers. Common and productive desktop tools include:

Mail Tool — Used to compose, view, and manage electronic mail through a GUI. Allows the inclusion of attachments and communications with other applications through the messaging system.

Calendar Manager — Used to manage, schedule, and view appointments, create calendars, and interact with the Mail Tool.

Editor — A text editor with common functionality including data transfer with other applications via the clipboard, drag and drop, and primary and quick transfer.

Terminal Emulator — An *xterm* terminal emulator.

Calculator — A standard calculator with scientific, financial, and logical modes.

Print Manager — A graphical print job manager for the scheduling and management of print jobs on any available printer.

Help System — A context-sensitive graphical help system based on Standard Generalized Markup Language (SGML).

Style Manager — A graphical interface that allows a user to interactively set their preferences, such as colors, backdrops, and fonts, for a session.

Icon Editor — This application is a fairly full featured graphical icon (pixmap) editor.

1.6 Application development tools

CDE includes two components for application development. The first is a shell command language interpreter that has built-in commands for most X Window system and CDE functions. The interpreter is based on ksh93 (The Korn Shell), and should provide anyone familiar with shell scripts the ability to develop X, Motif, and CDE applications.

To support interactive user interface development, developers can use the Motif Application Builder. This is a GUI front end for building Motif applications that generates C source code. The source code is then compiled and linked with the X and Motif libraries to produce the executable binary.

1.7 Application integration

CDE provides a number of tools to ease integration. The overall model of the CDE session is intended to allow a straightforward integration for virtually all types of applications. Motif and other X toolkit applications usually require little integration.

The task of integrating in-house and third party applications into a desktop, often the most difficult aspect of a desktop installation, is simplified by CDE. The power and advantage of CDE functionality can be realized in most cases without recompiling applications.

For example, Open Look applications can be integrated through the use of scripts that perform front-end execution of the application and scripts that perform pre- and post-session processing.

After the initial task of integrating applications so that they fit within session management, further integration can be done to increase their overall common *look-and-feel* with the rest of the desktop and to take advantage of the full range of CDE functionality. Tools that ease this aspect of integration include an *Icon Editor* used to create colour and monochrome icons. Images can be copied from the desktop into an icon, or they can be drawn freehand.

The *Action Creation Utility* is used to create action entries in the action database. Actions allow applications to be launched using desktop icons, and they ease administration by removing an application's specific details from the user interface.

The *Application Gather* and *Application Integrate* routines are used to control and format the application manager. They simplify installations so that applications can be accessible from virtually anywhere on the network.

1.8 Windows and the Window Manager

From a user's perspective, one of the first distinguishing features of Motif's *look and feel* is the *window frame* (Fig. 1.3). Every application window is contained inside such a frame. The following items appear in the window frame:

Title Bar — This identifies the window by a text string. The string is usually the name of the application program. However, an application's resource controls the label (Chapter 40).

Window Menu — Every window under the control of *mwm* has a window menu. The application has a certain amount of control over items that can be placed in the menu. The *Motif Style Guide* insists that certain commands are always available in this menu and that they can be accessed from either mouse or keyboard selection. Keyboard selections are called *mnemonics* and allow routine actions (that may involve several mouse actions) to be called from the keyboard. The action from the keyboard usually involves pressing two keys at the same time: the `Meta` key[1] and another key.

[1]The Meta key is an abstraction of the X Window System which is usually `alt` on

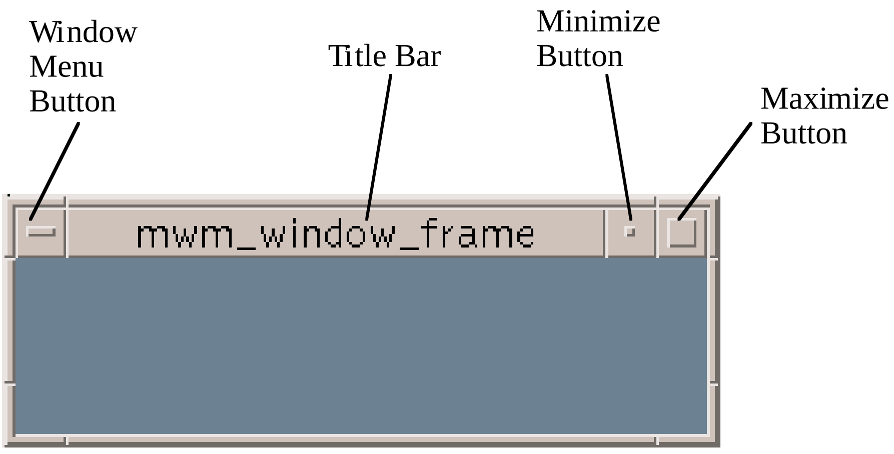


Figure 1.3: The Motif Window Frame

The default window menu items and *mnemonics* are listed below and illustrated in Fig. 1.4:

- **Restore** (`Meta+F5`) — Restore window to previous size after iconification (*see* below).
- **Move** (`Meta+F7`) — Allows the window to be repositioned with a drag of the mouse.
- **Size** (`Meta+F8`) — Allows the size of the window to be changed by dragging on the corners of the window.
- **Minimize** (`Meta+F9`) — Iconify the window.
- **Maximize** (`Meta+F10`) — Make the window the size of the root window, usually the whole of the display size.
- **Lower** (`Meta+F3`) — Move the window to the bottom of the window stack. Windows may be *tiled* on top of each other (*see* below). The front window being the top of the stack.
- **Close** (`Meta+F4`) — Quit the program. Some simple applications (Chapter 38) provide no *internal* means of termination. The `Close` option being the only means to achieve this.

most systems. However some systems may not posses such a key. Apple Macintoshes use the *Apple* key instead, for example. On Sun Type 4 keyboards the Meta key is the diamond shape key *next to* the `alt` key (*not* the `alt` key). Local X implementation should be consulted for further clarification. In this book we will simply refer to the `Meta` key.

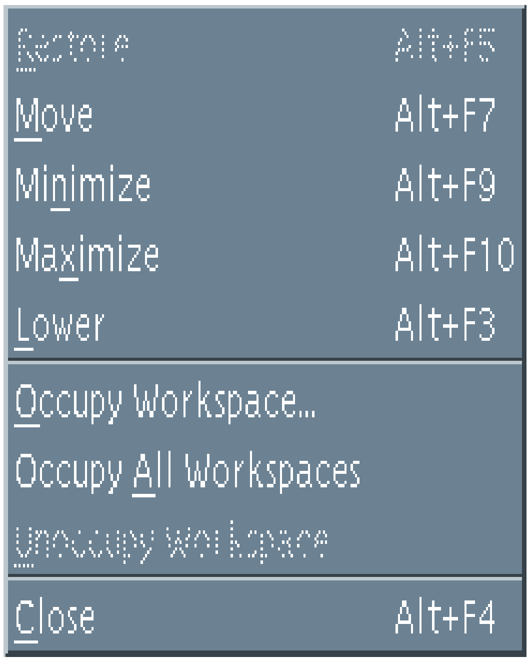


Figure 1.4: The Window Menu

Minimize Button — another way to iconify a window.

Maximize Button — another way to make a window the size of the root window.

The window manager must also be able to manage multiple windows from multiple client applications. There are a few important issues that need to be resolved. When running several applications together, several windows may be displayed on the screen. As a result, the display may appear cluttered and hard to navigate. The window manager provides two mechanisms to help deal with such problems:

Active Window — Only one window can receive input at any time. If you are selecting a graphical object with a mouse, then it is relatively easy for the window manager to detect this and schedule appropriate actions related to the chosen object. It is not so easy when you enter data or make selections directly from the keyboard. To resolve this only one window at a time is allowed *keyboard focus*. This window is called the *active window*. The selection of the active window will depend on the system configuration which the user typically has control over. There are two common methods for selecting the active window:

Focus follows pointer — The active window is the window is the window underneath mouse pointer.

Click-to-type — The active window is selected, by clicking on an area of the window, and remains active until another window is selected no matter where the mouse points.

When a window is made active its appearance will change slightly:

- Its outline frame will become shaded.
- The cursor will change appearance when placed in the window.
- The window may jump, or be *raised* to the top of the window stack.

The exact appearance of the above may vary from system to system and may be controlled by the user by setting environment settings in the window manager.

Window tiling — Windows may be stacked on top of each other. The window manager tries to maintain a three-dimensional *look and feel*. Apart from the fact that buttons, dialog boxes appear to be elevated from the screen, windows are shaded and framed in a three-dimensional fashion. The top window (or currently active window) will have slightly different appearance for instance.

The window menu has a few options for controlling the tiling of a window. Also a window can be brought to the top of the stack, or *raised* by clicking a part of its frame.

Iconification — If a window is currently active and not required for input or displaying output then it may be *iconified* or *minimised* thus reducing the screen clutter. An icon (Fig. 1.5) is a small graphical symbol that represents the window (or application). It occupies a significantly less amount of screen area. Icons are usually arranged around the perimeter (typically bottom or left side) of the screen. The application will still be running and occupying computer memory. The window related to the icon may be reverted to by either double clicking on the icon, or selecting *Restore* or *Maximise* from the icon's window menu.

Figure 1.5: Sample Icon from Xterm Application

1.9 The Root Menu

The *Root Menu* is the main menu of the window manager. The root menu typically is used to control the whole display, for example starting up new windows and quitting the desktop. To display the Root menu:

- Move the mouse pointer to the Root Window.
- Hold down the left mouse button.

The default Root Menu has the following The root menu can be customised to start up common applications for example. The root menu for the *mwm* (Fig. 1.6) and *dtwm* (Fig. 1.7) have slightly different appearance but have broadly similar actions, which are summarised below:

Program (*dtwm*) — A sub-menu is displayed that allows a variety of programs to be called from the desktop, for example to create a new window. The list of available programs can be customised from the desktop.

New Window (*mwm*) — Create a new window which is usually an *Xterm* window.

Shuffle Up — Move the bottom of the window stack to the top.

Shuffle Down — Move the top of the window stack to the bottom.

Refresh — Refresh the current screen display.

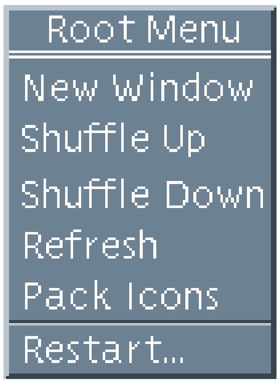


Figure 1.6: The *mwm* Root Menu

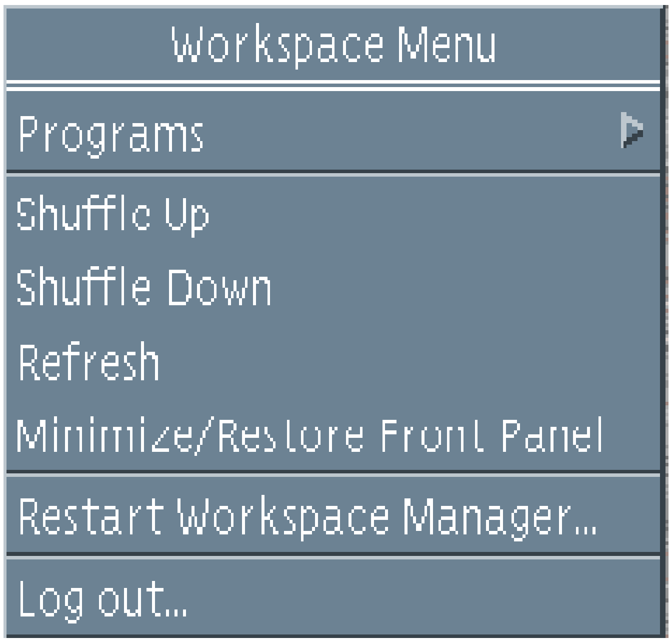


Figure 1.7: The CDE *dtwm* Root Menu

Restart — Restart the Workspace.

Logout (*dtwm*) — Quit the Window Manager.

1.10 Exercises

Exercise 1.1 *Add an application to the application manager*

Exercise 1.2 *Practice opening, closing and moving windows around the screen and to/from the background/foreground. Get used to using the mouse and its buttons for such tasks.*

Exercise 1.3 *Figure out the function of each of the three mouse buttons. Pay particular attention to the different functions the buttons in different windows (applications) and also when the mouse is pointing to the background.*

Exercise 1.4 *Find out how to resize windows etc. and practice this.*

Exercise 1.5 *Fire up the texteditor of your choice (You may use* `dtpad` *(basic but functional),* `textedit` *application (SOLARIS basic editor), emacs/Xemacs, or vi) and practice editing text files. Create any files you wish for now. Figure out basic options like cut and paste of text around the file, saving and loading files, searching for strings in the text and replacing strings.*

Particularly pay attention in getting used to using the Key Strokes and / or mouse to perform the above tasks.

Exercise 1.6 *Use Unix Commands to*

1. *Copy a file (created by text editor or other means) to another file called spare.*
2. *Rename your original file to one called new.*
3. *Delete the file spare.*
4. *Display your original file on the terminal.*
5. *Print your file out.*

Exercise 1.7 *Familiarise yourself with other UNIX functions by creating various files of text etc. and trying out the various functions listed in handouts.*

Chapter 2

C/C++ Program Compilation

In this chapter we begin by outlining the basic processes you need to go through in order to compile your C (or C++) programs. We then proceed to formally describe the C compilation model and also how C supports additional libraries.

2.1 Creating, Compiling and Running Your Program

The stages of developing your C program are as follows. (See Appendix A and exercises for more info.)

2.1.1 Creating the program

Create a file containing the complete program, such as the above example. You can use any ordinary editor with which you are familiar to create the file. One such editor is *textedit* available on most UNIX systems.

The filename must by convention end ".c" (full stop, lower case c), *e.g.* *myprog.c* or *progtest.c*. The contents must obey C syntax. For example, they might be as in the above example, starting with the line `/* Sample ....` (or a blank line preceding it) and ending with the line } `/* end of program */` (or a blank line following it).

2.1.2 Compilation

There are many C compilers around. The `cc` being the default Sun compiler. The GNU C compiler `gcc` is popular and available for many platforms. PC users may also be familiar with the Borland `bcc` compiler.

There are also equivalent C++ compilers which are usually denoted by `CC` (*note* upper case CC. For example Sun provides CC and GNU `GCC`. The GNU compiler is also denoted by `g++`

Other (less common) C/C++ compilers exist. All the above compilers operate in essentially the same manner and share many common command line options. Below and in Appendix A we list and give example uses many of the common compiler options. However, the **best** source of each compiler is through the online manual pages of your system: *e.g.* `man cc`.

For the sake of compactness in the basic discussions of compiler operation we will simply refer to the `cc` compiler — other compilers can simply be substituted in place of `cc` unless otherwise stated.

To Compile your program simply invoke the command `cc`. The command must be followed by the name of the (C) program you wish to compile. A number of compiler options can be specified also. We will not concern ourselves with many of these options yet, some useful and often essential options are introduced below — See Appendix A or online manual help for further details.

Thus, the basic compilation command is:

```
cc program.c
```

where *program.c* is the name of the file.

If there are obvious errors in your program (such as mistypings, misspelling one of the key words or omitting a semi-colon), the compiler will detect and report them.

There may, of course, still be logical errors that the compiler cannot detect. You may be telling the computer to do the wrong operations.

When the compiler has successfully digested your program, the compiled version, or executable, is left in a file called *a.out* or if the compiler option *-o* is used : the file listed after the *-o*.

It is more convenient to use a *-o* and filename in the compilation as in

```
cc -o program program.c
```

which puts the compiled program into the file program (or any file you name following the "-o" argument) **instead** of putting it in the file a.out .

2.1.3 Running the program

The next stage is to actually run your executable program. To run an executable in UNIX, you simply type the name of the file containing it, in this case *program* (or *a.out*)

This executes your program, printing any results to the screen. At this stage there may be run-time errors, such as division by zero, or it may become evident that the program has produced incorrect output.

If so, you must return to edit your program source, and recompile it, and run it again.

2.2 The C Compilation Model

We will briefly highlight key features of the C Compilation model (Fig. 2.1) here.

2.2.1 The Preprocessor

We will study this part of the compilation process in greater detail later (Chapter 13. However we need some basic information for some C programs.

The Preprocessor accepts source code as input and is responsible for

- removing comments
- interpreting special *preprocessor directives* denoted by `#`.

For example

- `#include` — includes contents of a named file. Files usually called *header* files. *e.g*
  - `#include <math.h>` — standard library maths file.
  - `#include <stdio.h>` — standard library I/O file
- `#define` — defines a symbolic name or constant. Macro substitution.
  - `#define MAX_ARRAY_SIZE 100`

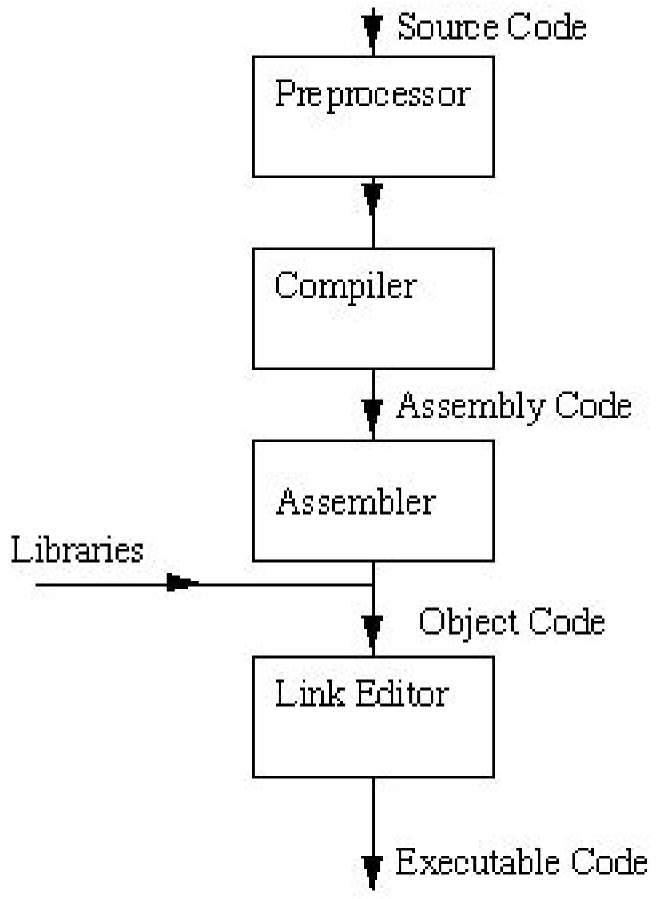


Figure 2.1: The C Compilation Model

2.2.2 C Compiler

The C compiler translates source to assembly code. The source code is received from the preprocessor.

2.2.3 Assembler

The assembler creates object code. On a UNIX system you may see files with a `.o` suffix (`.OBJ` on MSDOS) to indicate object code files.

2.2.4 Link Editor

If a source file references library functions or functions defined in other source files the *link editor* combines these functions (with `main()`) to create an executable file. External Variable references resolved here also. *More on this later* (Chapter 34).

2.2.5 Some Useful Compiler Options

Now that we have a basic understanding of the compilation model we can now introduce some useful and sometimes essential common compiler options. Again see the online `man` pages and Appendix A for further information and additional options.

`-c` Suppress the linking process and produce a `.o` file for each source file listed. Several can be subsequently linked by the `cc` command, for example:

```
cc file1.o file2.o ......  -o executable
```

`-llibrary` Link with object libraries. This option must follow the source file arguments. The object libraries are archived and can be standard, third party or user created libraries (We discuss this topic briefly below and also in detail later (Chapter 34). Probably the most commonly used library is the math library (`math.h`). You must link in this library explicitly if you wish to use the maths functions (**note** do note forget to `#include` <`math.h`> header file), for example:

```
cc calc.c -o calc -lm
```

Many other libraries are linked in this fashion (see below)

`-Ldirectory` Add directory to the list of directories containing object-library routines. The linker always looks for standard and other system libraries in `/lib` and `/usr/lib`. If you want to link in libraries that you have created or installed yourself (unless you have certain privileges and get the libraries installed in `/usr/lib`) you **will** have to specify where you files are stored, for example:

```
cc prog.c -L/home/myname/mylibs mylib.a
```

`-Ipathname` Add pathname to the list of directories in which to search for #include files with relative filenames (not beginning with slash /).

BY default, The preprocessor first searches for #include files in the directory containing source file, then in directories named with -I options (if any), and finally, in /usr/include. So to include header files stored in `/home/myname/myheaders` you would do:

```
cc prog.c -I/home/myname/myheaders
```

Note: System library header files are stored in a special place (`/usr/include`) and are not affected by the `-I option`. System header files and user header files are included in a slightly different manner (see Chapters 13 and 34)

`-g` invoke debugging option. This instructs the compiler to produce additional symbol table information that is used by a variety of debugging utilities.

`-D` define symbols either as identifiers (`-D`*identifer*) or as values (`-D`*symbol=value*) in a similar fashion as the `#define` preprocessor command. For more details on the use of this argument see Chapter 13.

For further information on general compiler options and the GNU compiler refer to Appendix A.

2.2.6 Using Libraries

C is an extremely small language. Many of the functions of other languages are not included in C. *e.g.* No built in I/O, string handling or maths functions.

What use is C then?

C provides functionality through a rich set function libraries.

As a result most C implementations include *standard* libraries of functions for many facilities (I/O *etc.*). For many practical purposes these may be regarded as being part of C. But they may vary from machine to machine. (*cf* Borland C for a PC to UNIX C).

A programmer can also develop his or her own function libraries and also include special purpose third party libraries (*e.g.* NAG, PHIGS).

All libraries (except standard I/O) need to be explicitly linked in with the `-l` and, possibly, `-L` compiler options described above.

2.2.7 UNIX Library Functions

The UNIX system provides a large number of C functions as libraries. Some of these implement frequently used operations, while others are very specialised in their application.

Do Not Reinvent Wheels: It is wise for programmers to check whether a library function is available to perform a task before writing their own version. This will reduce program development time. The library functions have been tested, so they are more likely to be correct than any function which the programmer might write. This will save time when debugging the program.

Later chapters deal with all important standard library issues and other common system libraries.

2.2.8 Finding Information about Library Functions

The UNIX manual has an entry for all available functions. Function documentation is stored in *section 3* of the manual, and there are many other useful system calls in *section 2*. If you already know the name of the function you want, you can read the page by typing (to find about `sqrt`):

```
man 3 sqrt
```

If you don't know the name of the function, a full list is included in the introductory page for section 3 of the manual. To read this, type

```
man 3 intro
```

There are approximately 700 functions described here. This number tends to increase with each upgrade of the system.

On any manual page, the SYNOPSIS section will include information on the use of the function. For example:

```
#include <time.h>

char *ctime(time_t *clock)
```

This means that you must have

```
#include <time.h>
```

in your file before you call `ctime`. And that function ctime takes a pointer to type `time_t` as an argument, and returns a `string (char *)`. `time_t` will probably be defined in the same manual page.

The DESCRIPTION section will then give a short description of what the function does. For example:

```
ctime() converts a long integer, pointed to by clock,  to  a
26-character  string  of the form produced by asctime().
```

2.3 Lint — A C program verifier

You will soon discover (if you have not already) that the C compiler is pretty vague in many aspects of checking program correctness, particularly in type checking. Careful use of prototyping of functions can assist modern C compilers in this task. However, There is still no guarantee that once you have successfully compiled your program that it will run correctly.

The UNIX utility `lint` can assist in checking for a multitude of programming errors. Check out the online manual pages (`man lint`) for complete details of lint. It is well worth the effort as it can help save many hours debugging your C code.

To run lint simply enter the command:

`lint myprog.c`.

Lint is particularly good at checking type checking of variable and function assignments, efficiency, unused variables and function identifiers, unreachable code and possibly memory leaks. There are many useful options to help control lint (see `man lint`).

2.4 Exercises

Exercise 2.1 *Enter, compile and run the following program:*

```
main()

{ int i;

   printf("\t Number \t\t Square of Number\n\n");

   for (i=0; i<=25;++i)
            printf("\t %d \t\t\t %d \n",i,i*i);

}
```

Exercise 2.2 *The following program uses the math library. Enter compile and run it correctly.*

```
#include <math.h>

main()

{ int i;

   printf("\t Number \t\t Square Root of Number\n\n");

   for (i=0; i<=360; ++i)
            printf("\t %d \t\t\t %d \n",i, sqrt((double) i));

}
```

Exercise 2.3 *Look in* `/lib` *and* `/usr/lib` *and see what libraries are available.*

- *Use the* `man` *command to get details of library functions*
- *Explore the libraries to see what each contains by running the command* `ar t libfile`*.*

Exercise 2.4 *Look in* `/usr/include` *and see what header files are available.*

- *Use the* `more` *or* `cat` *commands to view these text files*
- *Explore the header files to see what each contains, note the include, define, type definitions and function prototypes declared in them*

Exercise 2.5 *Suppose you have a C program whose main function is in* `main.c` *and has other functions in the files* `input.c` *and* `output.c`*:*

- *What command(s) would you use on your system to compile and link this program?*
- *How would you modify the above commands to link a library called* `process1` *stored in the standard system library directory?*
- *How would you modify the above commands to link a library called* `process2` *stored in your home directory?*
- *Some header files need to be read and have been found to located in a* `header` *subdirectory of your home directory and also in the current working directory. How would you modify the compiler commands to account for this?*

Exercise 2.6 *Suppose you have a C program composed of several separate files, and they include one another as shown below:*

File	*Include Files*
`main.c`	`stdio.h, process1.h`
`input.c`	`stdio.h, list.h`
`output.c`	`stdio.h`
`process1.c`	`stdio.h, process1.h`
`process2.c`	`stdio.h, list.h`

- *Which files have to recompiled after you make changes to* `process1.c`*?*
- *Which files have to recompiled after you make changes to* `process1.h`*?*
- *Which files have to recompiled after you make changes to* `list.h`*?*

Chapter 3

C Basics

Before we embark on a brief tour of C's basic syntax and structure we offer a brief history of C and consider the characteristics of the C language.

In the remainder of the Chapter we will look at the basic aspects of C programs such as C program structure, the declaration of variables, data types and operators. We will assume knowledge of a high level language, such as PASCAL.

It is our intention to provide a quick guide through similar C principles to most high level languages. Here the syntax may be slightly different but the concepts exactly the same.

C does have a few surprises:

- Many High level languages, like PASCAL, are highly disciplined and structured.

- **However beware** — C is much more flexible and free-wheeling. This freedom gives C much more **power** that experienced users can employ. The above example below (`mystery.c`) illustrates how bad things could really get.

3.1 History of C

The *milestones* in C's development as a language are listed below:

- UNIX developed c. 1969 — DEC PDP-7 Assembly Language

- BCPL — a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- A new language “B” a second attempt. c. 1970.
- A totally new language “C” a successor to “B”. c. 1971
- By 1973 UNIX OS almost totally written in “C”.

3.2 Characteristics of C

We briefly list some of C’s characteristics that define the language and also have lead to its popularity as a programming language. Naturally we will be studying many of these aspects throughout the course.

- Small size
- Extensive use of function calls
- Loose typing — unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

As an extreme example the following C code (`mystery.c`) is actually *legal* C code.

```
#include <stdio.h>

main(t,_,a)
char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+\
,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

It will compile and run and produce meaningful output. Try this program out. Try to compile and run it yourself.

Clearly nobody ever writes code like or at least should never. This piece of code actually one an international Obfuscated C Code Contest (see http://reality.sgi.com/csp/iocc)

The standard for C programs was originally the features set by Brian Kernighan. In order to make the language more internationally acceptable, an international standard was developed, ANSI C (American National Standards Institute).

3.3 C Program Structure

A C program basically has the following form:

- Preprocessor Commands
- Type definitions
- Function prototypes — declare function types and variables passed to function.
- Variables
- Functions

We must have a `main()` function.

A function has the form:

```
type function_name (parameters)
    {
      local variables

      C Statements

    }
```

If the type definition is omitted C assumes that function returns an **integer** type. **NOTE:** This can be a source of problems in a program.

So returning to our first C program:

```
/* Sample program */

main()
       {

       printf( ``I Like C \n'' );
       exit ( 0 );

       }
```

NOTE:

- C requires a semicolon at the end of **every** statement.
- `printf` is a *standard* C function — called from `main`.
- \n signifies newline. **Formatted output** — more later.
- `exit()` is also a standard function that causes the program to terminate. Strictly speaking it is not needed here as it is the last line of `main()` and the program will terminate anyway.

Let us look at another printing statement: `printf(``.\n.1\n..2\n...3\n'');`

The output of this would be:

```
.
.1
..2
...3
```

3.4 Variables

C has the following simple data types:

C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	−32768	+32767
unsigned short int	2	0	65536
(long) int	4	-2^{31}	$+2^{31}-1$
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

The Pascal Equivalents are:

C type	Pascal equivalent
char	char
unsigned char	—
short int	integer
unsigned short int	—
long int	longint
float	real
double	extended

On UNIX systems all `int`s are `long int`s unless specified as `short int` explicitly.

NOTE: There is **NO** Boolean type in C — you should use `char, int` or (better) `unsigned char`.

`Unsigned` can be used with all `char` and `int` types.

To declare a variable in C, do:
`var_type` *list variables*;

e.g. `int i,j,k;`

```
float x,y,z;
char ch;
```

3.4.1 Defining Global Variables

Global variables are defined above `main()` in the following way:-

```
short number,sum;
int bignumber,bigsum;
char letter;

main()
       {

       }
```

It is also possible to pre-initialise global variables using the `=` operator for assignment.

NOTE: The `=` operator is the same as `:=` is Pascal.

For example:-

```
float sum=0.0;
int bigsum=0;
char letter='A';

main()
       {

       }
```

This is the same as:-

```
float sum;
int bigsum;
char letter;
```

```
main()
       {

       sum=0.0;
       bigsum=0;
       letter='A';

       }
```

...but is more efficient.

C also allows multiple assignment statements using =, for example:

```
a=b=c=d=3;
```

...which is the same as, but more efficient than:

```
a=3;
b=3;
c=3;
d=3;
```

This kind of assignment is only possible if all the variable types in the statement are the same.

You can define your own types use `typedef`. This will have greater relevance later in the course when we learn how to create more complex data structures.

As an example of a simple use let us consider how we may define two new types `real` and `letter`. These new types can then be used in the same way as the pre-defined C types:

```
typedef float real;
typedef char letter;
```

Variables declared:

```
real sum=0.0;
letter nextletter;
```

3.4.2 Printing Out and Inputting Variables

C uses formatted output. The `printf` function has a special formatting character (%) — a character following this defines a certain format for a variable:

%c — characters
%d — integers
%f — floats

e.g. `printf(‘‘%c %d %f’’,ch,i,x);`

NOTE: Format statement enclosed in “...”, variables follow after. Make sure order of format and variable data types match up.

`scanf()` is the function for inputting values to a data structure: Its format is similar to `printf`:

i.e. `scanf(‘‘%c %d %f’’,&ch,&i,&x);`

NOTE: & before variables. Please accept this for now and **remember** to include it. It is to do with pointers which we will meet later (Section 17.4.1).

3.5 Constants

ANSI C allows you to declare *constants*. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.

The `const` keyword is to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

Note:

- You can declare the `const` before or after the type. Choose one an stick to it.
- It is usual to initialise a `const` with a value as it cannot get a value *any other way*.

The preprocessor `#define` is another more flexible (see Preprocessor Chapters) method to define *constants* in a program.

You frequently see const declaration in function parameters. This says simply that the function is **not** going to change the value of the parameter.

The following function definition used concepts we have not met (see chapters on functions, strings, pointers, and standard libraries) but for completenes of this section it is is included here:

```
void strcpy(char *buffer, char const *string)
```

The second argiment `string` is a C string that will not be altered by the string copying standard library function.

3.6 Arithmetic Operations

As well as the standard arithmetic operators ($+ \quad - \quad * \quad /$) found in most languages, C provides some more operators. There are some notable differences with other languages, such as Pascal.

Assignment is = *i.e.* $i = 4$; `ch` = 'y';

Increment ++, Decrement −− which are more efficient than their long hand equivalents, for example:- `x++` is faster than `x=x+1`.

The ++ and −− operators can be either in post-fixed or pre-fixed. With pre-fixed the value is computed before the expression is evaluated whereas with post-fixed the value is computed after the expression is evaluated.

In the example below, `++z` is pre-fixed and the `w--` is post-fixed:

```
int x,y,w;

main()
       {

       x=((++z)-(w--)) % 100;

       }
```

This would be equivalent to:

```
     int x,y,w;

     main()
            {

            z++;
            x=(z-w) % 100;
            w--;

            }
```

The % (modulus) operator only works with integers.

Division / is for both integer and float division. So be careful.

The answer to: $x = 3/2$ is 1 even if x is declared a float!!

RULE: If both arguments of / are integer then do integer division.

So make sure you do this. The correct (for division) answer to the above is $x = 3.0/2$ or $x = 3/2.0$ or (better) $x = 3.0/2.0$.

There is also a convenient **shorthand** way to express computations in C.

It is very common to have expressions like: $i = i + 3$ or $x = x * (y + 2)$

This can written in C (generally) in a *shorthand* form like this:
$expr_1 \; op \; = \; expr_2$

which is equivalent to (but more efficient than):
$expr_1 \; = \; expr_1 \; op \; expr_2$

So we can rewrite $i = i + 3$ as $i+ = 3$

and $x = x * (y + 2)$ as $x* = y + 2$.

NOTE: that $x* = y + 2$ means $x = x * (y + 2)$ and **<u>NOT</u>** $x = x * y + 2$.

3.7 Comparison Operators

To test for equality is ==

A warning: Beware of using "=" instead of "==", such as writing accidentally

```
if ( i = j ) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if `j` is non-zero. This is called **assignment by value** — a key feature of C.

Not equals is: ! =

Other operators < (less than) , > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

3.8 Logical Operators

Logical operators are usually used with conditional statements which we shall meet in the next Chapter.

The two basic logical operators are:

&& for logical AND, || for logical OR.

Beware & and | have a different meaning for bitwise AND and OR (*more on this later* in Chapter 12).

3.9 Order of Precedence

It is necessary to be careful of the meaning of such expressions as `a + b * c`

We may want the effect as either

```
(a + b) * c
```

or

`a + (b * c)` All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that

```
a - b - c
```

is evaluated as

```
( a - b ) - c
```

as you would expect.

From high priority to low priority the order for all C operators (we have not met all of them yet) is:

( ) [ ] − > .
! ∼ − ∗ & `sizeof` `cast` + + − −
    (these are right->left)
∗ / %
+ −
< <= >= >
== ! =
&
∧
|
&&
||
?: (right->left)
= + = − = (right->left)
, (comma)

Thus

```
  a < 10 && 2 * b < c
```

is interpreted as `( a < 10 ) && ( ( 2 * b ) < c )`

and

```
a =

  b =
    spokes / spokes_per_wheel
    + spares;
```

as

```
a =

  ( b =
      ( spokes / spokes_per_wheel )
      + spares
  );
```

3.10 Exercises

Write C programs to perform the following tasks.

Exercise 3.1 *Input two numbers and work out their sum, average and sum of the squares of the numbers.*

Exercise 3.2 *Input and output your name, address and age to an appropriate structure.*

Exercise 3.3 *Write a program that works out the largest and smallest values from a set of 10 inputted numbers.*

Exercise 3.4 *Write a program to read a "float" representing a number of degrees Celsius, and print as a "float" the equivalent temperature in degrees Fahrenheit. Print your results in a form such as*
100.0 degrees Celsius converts to 212.0 degrees Fahrenheit.

Exercise 3.5 *Write a program to print several lines (such as your name and address). You may use either several printf instructions, each with a newline character in it, or one printf with several newlines in the string.*

Exercise 3.6 *Write a program to read a positive integer at least equal to 3, and print out all possible permutations of three positive integers less or equal to than this value.*

Exercise 3.7 *Write a program to read a number of units of length (a float) and print out the area of a circle of that radius. Assume that the value of pi is 3.14159 (an appropriate declaration will be given you by ceilidh – select setup).*

Your output should take the form: The area of a circle of radius ... units is .... units.

If you want to be clever, and have looked ahead in the notes, print the message Error: Negative values not permitted. if the input value is negative.

Exercise 3.8 *Given as input a floating (real) number of centimeters, print out the equivalent number of feet (integer) and inches (floating, 1 decimal), with the inches given to an accuracy of one decimal place.*
Assume 2.54 centimeters per inch, and 12 inches per foot.
If the input value is 333.3, the output format should be:
333.3 centimeters is 10 feet 11.2 inches.

Exercise 3.9 *Given as input an integer number of seconds, print as output the equivalent time in hours, minutes and seconds. Recommended output format is something like*

7322 seconds is equivalent to 2 hours 2 minutes 2 seconds.

Exercise 3.10 *Write a program to read two integers with the following significance.*

The first integer value represents a time of day on a 24 hour clock, so that 1245 represents quarter to one mid-day, for example.

The second integer represents a time duration in a similar way, so that 345 represents three hours and 45 minutes.

This duration is to be added to the first time, and the result printed out in the same notation, in this case 1630 which is the time 3 hours and 45 minutes after 12.45.

Typical output might be Start time is 1415. Duration is 50. End time is 1505.

There are a few extra marks for spotting.

Start time is 2300. Duration is 200. End time is 100.

Chapter 4

Conditionals

This Chapter deals with the various methods that C can control the *flow* of logic in a program. Apart from slight syntactic variation they are similar to other languages.

As we have seen following logical operations exist in C:

==, ! =, ||, &&.

One other operator is the unitary – it takes only one argument – *not* **!**.

These operators are used in conjunction with the following statements.

4.1 The `if` statement

The `if` statement has the same function as other languages. It has three basic forms:

`if` (*expression*)
    statement

...or:

`if` (*expression*)
    statement$_1$
`else`
    statement$_2$

...or:

```
if  (expression)
    statement_1
else if (expression)
    statement_2
else
    statement_3
```

For example:-

```
int x,y,w;

main()
        {

        if (x>0)
           {
           z=w;
           ........
           }
        else
           {
           z=y;
           ........
           }

        }
```

4.2 The `?` operator

The `?` (*ternary condition*) operator is a more efficient form for expressing simple `if` statements. It has the following form:

expression$_1$ `?` *expression*$_2$ `:` *expression*$_3$

It simply states:

`if` *expression*$_1$ `then` *expression*$_2$ `else` *expression*$_3$

For example to assign the maximum of `a` and `b` to `z`:

```
z = (a>b) ?  a :  b;
```

which is the same as:

```
if (a>b)
   z = a;
else
   z=b;
```

4.3 The `switch` statement

The C `switch` is similar to Pascal's `case` statement and it allows multiple choice of a selection of items at one level of a conditional where it is a far neater way of writing multiple `if` statements:

```
switch (expression) {
        case item_1:
             statement_1;
             break;
        case item_2:
             statement_2;
             break;
             ⋮
             ⋮
        case item_n:
             statement_n;
             break;
        default:
             statement;
             break;
        }
```

In each case the value of *item*$_i$ must be a constant, variables are not allowed.

The `break` is needed if you want to terminate the `switch` after execution of one choice. Otherwise the next case would get evaluated. **Note:** This is unlike most other languages.

We can also have **null** statements by just including a ; or let the switch statement *fall through* by omitting any statements (see *e.g.* below).

The `default` case is optional and catches any other cases.

For example:-

```
switch (letter)
       {
       case ‘A’:
       case ‘E’:
       case ‘I’:
       case ‘O’:
       case ‘U’:
            numberofvowels++;
            break;

       case ‘ ’:
            numberofspaces++;
            break;

       default:
            numberofconstants++;
            break;
       }
```

In the above example if the value of `letter` is ‘A’, ‘E’, ‘I’, ‘O’ or ‘U’ then `numberofvowels` is incremented.

If the value of `letter` is ‘ ’ then `numberofspaces` is incremented.

If none of these is true then the `default` condition is executed, that is `numberofconstants` is incremented.

4.4 Exercises

Exercise 4.1 *Write a program to read two characters, and print their value when interpreted as a 2-digit hexadecimal number. Accept upper case letters for values from 10 to 15.*

Exercise 4.2 *Read an integer value. Assume it is the number of a month of the year; print out the name of that month.*

Exercise 4.3 *Given as input three integers representing a date as day, month, year, print out the number day, month and year for the following day's date.*

Typical input: 28 2 1992 Typical output: Date following 28:02:1992 is 29:02:1992

Exercise 4.4 *Write a program which reads two integer values. If the first is less than the second, print the message up. If the second is less than the first, print the message down If the numbers are equal, print the message equal If there is an error reading the data, print a message containing the word Error and perform exit(0);*

Chapter 5

Looping and Iteration

This chapter will look at C's mechanisms for controlling looping and iteration. Even though some of these mechanisms may look familiar and indeed will operate in standard fashion most of the time. **NOTE:** some non-standard features are available.

5.1 The `for` statement

The C `for` statement has the following form:

`for` (*expression*$_1$; *expression*$_2$; *expression*$_3$)
 statement;
 or {*block of statements*}

expression$_1$ initialises; *expression*$_2$ is the terminate test; *expression*$_3$ is the modifier (which may be more than just simple increment);

NOTE: C basically treats `for` statements as `while` type loops

For example:

```
int x;

main()
        {
        for (x=3;x>0;x--)
```

```
              {
              printf("x=%d\n",x);
              }
          }
```

...outputs:

```
x=3
x=2
x=1
```

...to the screen

All the following are legal `for` statements in C. The practical application of such statements is not important here, we are just trying to illustrate peculiar features of C `for` that may be useful:-

```
for (x=0;((x>3) && (x<9)); x++)

for (x=0,y=4;((x>3) && (y<9)); x++,y+=2)

for (x=0,y=4,z=4000;z; z/=10)
```

The second example shows that multiple expressions can be separated a `,`.

In the third example the loop will continue to iterate until `z` becomes 0;

5.2 The `while` statement

The `while` statement is similar to those used in other languages although more can be done with the `expression` statement — a standard feature of C.

The `while` has the form:

`while (`*expression*`)`
        statement

For example:

```
int x=3;

main()
       { while (x>0)
               { printf("x=%d\n",x);
                 x--;
               }
       }
```

...outputs:

```
x=3
x=2
x=1
```

...to the screen.

Because the `while` loop can accept expressions, not just conditions, the following are all legal:-

```
while (x--);
while (x=x+1);
while (x+=5);
```

Using this type of expression, only when the result of `x−−`, `x=x+1`, or `x+ = 5`, evaluates to 0 will the `while` condition fail and the loop be exited.

We can go further still and perform complete operations within the `while` *expression*:

```
while (i++ < 10);

while ( (ch = getchar()) != 'q')
  putchar(ch);
```

The first example counts `i` up to 10.

The second example uses C standard library functions (See Chapter 18) `getchar()` – reads a character from the keyboard – and `putchar()` – writes a given char to screen. The `while` loop will proceed to read from the keyboard and echo characters to the screen until a 'q' character is read. **NOTE:** This type of operation is used a lot in C and not just with character reading!! (See Exercises).

5.3 The `do-while` statement

C's `do-while` statement has the form:

```
do
    statement;
    while (expression);
```

It is similar to PASCAL's `repeat` ... `until` except `do while` *expression* is true.

For example:

```
int x=3;

main()
       { do {
            printf("x=%d\n",x--);
            }
       while (x>0);
       }
```

..outputs:-

```
x=3
x=2
x=1
```

NOTE: The postfix `x--` operator which uses the current value of `x` while printing and *then* decrements `x`.

5.4 break and continue

C provides two commands to control how we loop:

- `break` — exit form loop or switch.
- `continue` — skip 1 iteration of loop.

Consider the following example where we read in integer values and process them according to the following conditions. If the value we have read is negative, we wish to print an error message and abandon the loop. If the value read is great than 100, we wish to ignore it and continue to the next value in the data. If the value is zero, we wish to terminate the loop.

```
while (scanf( “%d”, &value ) == 1 && value ! = 0) {

      if (value < 0) {
         printf(“Illegal value\n”);
         break;
         /* Abandon the loop */
      }

      if (value > 100) {
         printf(“Invalid value\n”);
         continue;
         /* Skip to start loop again */
      }

      /* Process the value read */
      /* guaranteed between 1 and 100 */
      ....;

      ....;
} /* end while value ! = 0 */
```

5.5 Exercises

Exercise 5.1 *Write a program to read in 10 numbers and compute the average, maximum and minimum values.*

Exercise 5.2 *Write a program to read in numbers until the number -999 is encountered. The sum of all number read until this point should be printed out.*

Exercise 5.3 *Write a program which will read an integer value for a base, then read a positive integer written to that base and print its value.*

Read the second integer a character at a time; skip over any leading non-valid (i.e. not a digit between zero and "base-1") characters, then read valid characters until an invalid one is encountered.

```
          Input        Output
        ==========     ======
        10    1234      1234
         8      77        63   (the value of 77 in base 8, octal)
         2    1111        15   (the value of 1111 in base 2, binary)
```

The base will be less than or equal to 10.

Exercise 5.4 *Read in three values representing respectively*

a capital sum (integer number of pence),

a rate of interest in percent (float),

and a number of years (integer).

Compute the values of the capital sum with compound interest added over the given period of years. Each year's interest is calculated as

*interest = capital * interest_rate / 100;*

and is added to the capital sum by

capital += interest;

Print out money values as pounds (pence / 100.0) accurate to two decimal places.

Print out a floating value for the value with compound interest for each year up to the end of the period.

Print output year by year in a form such as:

```
Original sum 30000.00 at  12.5 percent for 20 years

Year Interest  Sum
----+-------+--------
  1  3750.00 33750.00
  2  4218.75 37968.75
  3  4746.09 42714.84
  4  5339.35 48054.19
  5  6006.77 54060.96
  6  6757.62 60818.58
  7  7602.32 68420.90
  8  8552.61 76973.51
  9  9621.68 86595.19
 10 10824.39 97419.58
```

Exercise 5.5 *Read a positive integer value, and compute the following sequence: If the number is even, halve it; if it's odd, multiply by 3 and add 1. Repeat this process until the value is 1, printing out each value. Finally print out how many of these operations you performed.*

Typical output might be:

```
Inital value is 9
Next value is  28
Next value is  14
Next value is   7
Next value is  22
Next value is  11
Next value is  34
Next value is  17
Next value is  52
Next value is  26
Next value is  13
Next value is  40
Next value is  20
Next value is  10
Next value is   5
Next value is  16
Next value is   8
Next value is   4
Next value is   2
Final value 1, number of steps 19
```

If the input value is less than 1, print a message containing the word

```
      Error
```

and perform an

```
        exit( 0 );
```

Exercise 5.6 *Write a program to count the vowels and letters in free text given as standard input. Read text a character at a time until you encounter end-of-data.*

Then print out the number of occurrences of each of the vowels a, e, i, o and u in the text, the total number of letters, and each of the vowels as an integer percentage of the letter total.

Suggested output format is:

```
        Numbers of characters:
        a   3 ; e   2 ; i   0 ; o   1 ; u   0 ; rest  17
        Percentages of total:
        a  13%; e   8%; i   0%; o   4%; u   0%; rest  73%
```

Read characters to end of data using a construct such as

```
        char ch;
        while(
            ( ch = getchar() ) >= 0
        ) {
            /* ch is the next character */    ....
        }
```

to read characters one at a time using `getchar()` *until a negative value is returned.*

Exercise 5.7 *Read a file of English text, and print it out one word per line, all punctuation and non-alpha characters being omitted.*

For end-of-data, the program loop should read until "getchar" delivers a value ¡= 0. When typing input, end the data by typing the end-of-file character, usually control-D. When reading from a file, "getchar" will deliver a negative value when it encounters the end of the file.

Typical output might be

```
Read
a
file
```

```
of
English
text
and
print
it
out
one
```

etc.

Chapter 6

Arrays and Strings

In principle arrays in C are similar to those found in other languages. As we shall shortly see arrays are defined slightly differently and there are many subtle differences due the close link between array and pointers. We will look more closely at the link between pointer and arrays later in Chapter 9.

6.1 Single and Multi-dimensional Arrays

Let us first look at how we define arrays in C:

```
int listofnumbers[50];
```

BEWARE: In C Array subscripts start at **0** and end one less than the array size. For example, in the above case valid subscripts range from 0 to 49. This is a **BIG** difference between C and other languages and does require a bit of practice to get in *the right frame of mind.*

Elements can be accessed in the following ways:-

```
thirdnumber=listofnumbers[2];
listofnumbers[5]=100;
```

Multi-dimensional arrays can be defined as follows:

```
int tableofnumbers[50][50];
```

for two dimensions.
For further dimensions simply add more [ ]:

```
int bigD[50][50][40][30]......[50];
```

Elements can be accessed in the following ways:

```
anumber=tableofnumbers[2][3];
tableofnumbers[25][16]=100;
```

6.2 Strings

In C Strings are defined as arrays of characters. For example, the following defines a string of 50 characters:

```
char name[50];
```

C has no string handling facilities built in and so the following are all illegal:

```
char firstname[50],lastname[50],fullname[100];

firstname= "Arnold"; /* Illegal */
lastname= "Schwarznegger"; /* Illegal */
fullname= "Mr"+firstname
          +lastname; /* Illegal */
```

However, there is a special library of string handling routines which we will come across later.

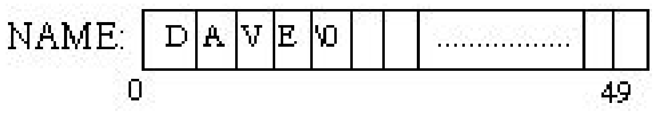
To print a string we use printf with a special **%s** control character:

`printf(‘‘%s’’,name);`

NOTE: We just need to give the name of the string.

In order to allow variable length strings the \0 character is used to indicate the end of a string.

So we if we have a string, `char NAME[50];` and we store the “DAVE” in it its contents will look like:



6.3 Exercises

Exercise 6.1 *Write a C program to read through an array of any type. Write a C program to scan through this array to find a particular value.*

Exercise 6.2 *Read ordinary text a character at a time from the program's standard input, and print it with each line reversed from left to right. Read until you encounter end-of-data (see below).*

You may wish to test the program by typing

```
prog5rev | prog5rev
```

to see if an exact copy of the original input is recreated.

To read characters to end of data, use a loop such as either

```
char ch;
while( ch = getchar(), ch >= 0 ) /* ch < 0 indicates end-of-data */
```

or

```
char ch;
while( scanf( "%c", &ch ) == 1 ) /* one character read */
```

Exercise 6.3 *Write a program to read English text to end-of-data (type control-D to indicate end of data at a terminal, see below for detecting it), and print a count of word lengths, i.e. the total number of words of length 1 which occurred, the number of length 2, and so on.*

Define a word to be a sequence of alphabetic characters. You should allow for word lengths up to 25 letters.

Typical output should be like this:

```
  length 1 : 10 occurrences
  length 2 : 19 occurrences
   length 3 : 127 occurrences
  length 4 : 0 occurrences
  length 5 : 18 occurrences
 ....
```

To read characters to end of data see above question.

Chapter 7

Functions

C provides functions which are again similar most languages. One difference is that C regards `main()` as function. Also unlike some languages, such as Pascal, C does not have *procedures* — it uses functions to service both requirements.

Let us remind ourselves of the form of a function:

returntype `fn_name(`*parameterdef*$_1$`,` *parameterdef*$_2$`,` $\cdots$`)`

{

localvariables

functioncode

}

Let us look at an example to find the average of two integers:

```
float findaverage(float a, float b)
      { float average;
        average=(a+b)/2;
        return(average);
      }
```

We would *call* the function as follows:

```
main()
      { float a=5,b=15,result;

        result=findaverage(a,b);
        printf("average=%f\n",result);
      }
```

Note: The `return` statement passes the result back to the main program.

7.1 `void` functions

The `void` function provide a way of emulating PASCAL type procedures.

If you do not want to return a value you must use the return type `void` and miss out the `return` statement:

```
void squares()
     { int loop;

       for (loop=1;loop<10;loop++);
           printf("%d\n",loop*loop);
     }

main()

     { squares();
     }
```

NOTE: We must have () even for no parameters unlike some languages.

7.2 Functions and Arrays

Single dimensional arrays can be passed to functions as follows:-

```
float findaverage(int size,float list[])

      { int i;
        float sum=0.0;

        for (i=0;i<size;i++)
            sum+=list[i];
        return(sum/size);
      }
```

Here the declaration `float list[]` tells C that `list` is an array of `float`. **Note** we do not specify the dimension of the array when it is a *parameter* of a function.

Multi-dimensional arrays can be passed to functions as follows:

```
void printtable(int xsize,int ysize,
        float table[][5])

     { int x,y;

       for (x=0;x<xsize;x++)
           { for (y=0;y<ysize;y++)
               printf("\t%f",table[x][y]);
             printf("\n");
       }
     }
```

Here `float table[][5]` tells C that `table` is an array of dimension N×5 of `float`. **Note** we must specify the second (and subsequent) dimension of the array BUT not the first dimension.

7.3 Function Prototyping

Before you use a function C must have *knowledge* about the type it returns and the parameter types the function expects.

The ANSI standard of C introduced a new (better) way of doing this than previous versions of C. (Note: All new versions of C now adhere to the ANSI standard.)

The importance of prototyping is twofold.

- It makes for more structured and therefore easier to read code.
- It allows the C compiler to check the *syntax* of function calls.

How this is done depends on the scope of the function (See Chapter 34). Basically if a functions has been defined before it is used (called) then you are ok to merely use the function.

If NOT then you must *declare* the function. The declaration simply states the type the function returns and the type of parameters used by the function.

It is usual (and therefore **good**) practice to prototype all functions at the start of the program, although this is not strictly necessary.

To *declare* a function prototype simply state the type the function returns, the function name and in brackets list the type of parameters in the order they appear in the function definition.

e.g.

```
int strlen(char []);
```

This states that a function called `strlen` returns an integer value and accepts a single string as a parameter.

NOTE: Functions can be prototyped and variables defined on the same line of code. This used to be more popular in pre-ANSI C days since functions are usually prototyped separately at the start of the program. This is still perfectly legal though: order they appear in the function definition.

e.g.

```
int length, strlen(char []);
```

Here `length` is a variable, `strlen` the function as before.

7.4 Exercises

Exercise 7.1 *Write a function "replace" which takes a pointer to a string as a parameter, which replaces all spaces in that string by minus signs, and delivers the number of spaces it replaced.*

Thus

```
        char *cat = "The cat sat";
        n = replace( cat );
```

should set

```
        cat to "The-cat-sat"
```

and

```
        n to 2.
```

Exercise 7.2 *Write a program which will read in the source of a C program from its standard input, and print out all the starred items in the following statistics for the program (all as integers). (Note the comment on tab characters at the end of this specification.)*

Print out the following values:

```
  Lines:
  *  The total number of lines
  *  The total number of blank lines
        (Any lines consisting entirely of white space should be
        considered as blank lines.)
     The percentage of blank lines (100 * blank_lines / lines)

  Characters:
  *  The total number of characters after tab expansion
  *  The total number of spaces after tab expansion
  *  The total number of leading spaces after tab expansion
       (These are the spaces at the start of a line, before any visible
        character; ignore them if there are no visible characters.)
    The average number of
      characters per line
      characters per line ignoring leading spaces
      leading spaces per line
      spaces per line ignoring leading spaces

  Comments:
  *  The total number of comments in the program
  *  The total number of characters in the comments in the program
       excluding the "/*" and "*/" thenselves
    The percentage of number of comments to total lines
    The percentage of characters in comments to characters

  Identifiers:
    We are concerned with all the occurrences of "identifiers" in the
      program where each part of the text starting with a letter,
      and continuing with letter, digits and underscores is considered
      to be an identifier, provided that it is not
          in a comment,
          or in a string,
          or within primes.
        Note that
            "abc\"def"
        the internal escaped quote does not close the string.
        Also, the representation of the escape character is
            '\\'
 and of prime is
            '\''
      Do not attempt to exclude the fixed words of the language,
      treat them as identifiers. Print
  *  The total number of identifier occurrences.
  *  The total number of characters in them.
    The average identifier length.

  Indenting:
  *  The total number of times either of the following occurs:
      a line containing a "}" is more indented than the preceding line
      a line is preceded by a line containing a "{" and is less
        indented than it.
      The "{" and "}" must be ignored if in a comment or string or
        primes, or if the other line involved is entirely comment.
```

```
    A single count of the sum of both types of error is required.
```

NOTE: All tab characters ('̂') on input should be interpreted as multiple spaces using the rule:

```
  "move to the next modulo 8 column"
  where the first column is numbered column 0.
 col before tab | col after tab
        ---------------+--------------
             0      |      8
             1      |      8
             7      |      8
             8      |     16
             9      |     16
            15      |     16
            16      |     24
```

To read input a character at a time the skeleton has code incorporated to read a line at a time for you using

```
char ch;
ch = getchar();
```

Which will deliver each character exactly as read. The "getline" function then puts the line just read in the global array of characters "linec", null terminated, and delivers the length of the line, or a negative value if end of data has been encountered.

You can then look at the characters just read with (for example)

```
switch( linec[0] ) {
case ' ': /* space ..... */
        break;
case '\t': /* tab character .... */
        break;
case '\n': /* newline ... */
        break;
....
} /* end switch */
```

End of data is indicated by scanf NOT delivering the value 1.

Your output should be in the following style:

```
Total lines                    126
Total blank lines              3
Total characters               3897
Total spaces                   1844
Total leading spaces           1180
Total comments                 7
Total chars in comments        234
Total number of identifiers    132
Total length of identifiers    606
Total indenting errors         2
```

You may gather that the above program (together with the unstarred items) forms the basis of part of your marking system! Do the easy bits first, and leave it at that if some aspects worry you. Come back to me if you think my solution (or the specification) is wrong! That is quite possible!

Exercise 7.3 *It's rates of pay again!*
Loop performing the following operation in your program:

Read two integers, representing a rate of pay (pence per hour) and a number of hours. Print out the total pay, with hours up to 40 being paid at basic rate, from 40 to 60 at rate-and-a-half, above 60 at double-rate. Print the pay as pounds to two decimal places.

Terminate the loop when a zero rate is encountered. At the end of the loop, print out the total pay.
The code for computing the pay from the rate and hours is to be written as a function.
The recommended output format is something like:

```
Pay at 200 pence/hr for 38 hours is 76.00 pounds
Pay at 220 pence/hr for 48 hours is 114.40 pounds
Pay at 240 pence/hr for 68 hours is 206.40 pounds
Pay at 260 pence/hr for 48 hours is 135.20 pounds
Pay at 280 pence/hr for 68 hours is 240.80 pounds
Pay at 300 pence/hr for 48 hours is 156.00 pounds
Total pay is 928.80 pounds
```

The "program features" checks that explicit values such as 40 and 60 appear only once, as a `#define` *or initialised variable value. This represents good programming practice.*

Chapter 8

Further Data Types

This Chapter discusses how more advanced data types and structures can be created and used in a C program.

8.1 Structures

Structures in C are similar to records in Pascal. For example:

```
struct gun
       {
       char name[50];
       int magazinesize;
       float calibre;
       };

struct gun arnies;
```

defines a new structure `gun` and makes `arnies` an instance of it.

NOTE: that `gun` is a *tag* for the structure that serves as shorthand for future declarations. We now only need to say `struct gun` and the body of the structure is implied as we do to make the `arnies` variable. The tag is *optional*.

Variables can also be declared between the } and ; of a struct declaration, *i.e.*:

```
  struct gun
         {
         char name[50];
         int magazinesize;
         float calibre;
         } arnies;
```

`struct`'s can be pre-initialised at declaration:

```
struct gun arnies={"Uzi",30,7};
```

which gives `arnie` a 7mm. Uzi with 30 rounds of ammunition.

To access a member (or field) of a `struct`, C provides the `.` operator. For example, to give `arnie` more rounds of ammunition:

```
arnies.magazineSize=100;
```

8.1.1 Defining New Data Types

`typedef` can also be used with structures. The following creates a new type `agun` which is of type `struct gun` and can be initialised as usual:

```
  typedef struct gun
                 {
                 char name[50];
                 int magazinesize;
                 float calibre;
                 } agun;

  agun arnies={"Uzi",30,7};
```

Here `gun` still acts as a *tag* to the `struct` and is optional. Indeed since we have defined a new data type it is not really of much use,

`agun` is the new data type. `arnies` is a variable of type `agun` which is a structure.

C also allows arrays of structures:

```
typedef struct gun
               {
               char name[50];
               int magazinesize;
               float calibre;
               } agun;

agun arniesguns[1000];
```

This gives `arniesguns` a 1000 guns. This may be used in the following way:

```
        arniesguns[50].calibre=100;
```

gives Arnie's gun number 50 a calibre of 100mm, and:

```
        itscalibre=arniesguns[0].calibre;
```

assigns the calibre of Arnie's first gun to `itscalibre`.

8.2 Unions

A union is a variable which may hold (at different times) objects of different sizes and types. C uses the `union` statement to create unions, for example:

```
        union number
              {
              short shortnumber;
              long longnumber;
              double floatnumber;
              } anumber
```

defines a union called `number` and an instance of it called `anumber`. `number` is a union *tag* and acts in the same way as a tag for a structure.

Members can be accessed in the following way:

```
        printf("%ld\n",anumber.longnumber);
```

This clearly displays the value of `longnumber`.

When the C compiler is allocating memory for unions it will always reserve enough room for the largest member (in the above example this is 8 bytes for the `double`).

In order that the program can keep track of the type of union variable being used at a given time it is common to have a structure (with union embedded in it) and a variable which flags the union type:

An example is:

```
typedef struct
        { int maxpassengers;
        } jet;

typedef struct
        { int liftcapacity;
        } helicopter;

typedef struct
        { int maxpayload;
        } cargoplane;

typedef union
        { jet jetu;
          helicopter helicopteru;
          cargoplane cargoplaneu;
        } aircraft;

typedef struct
        { aircrafttype kind;
          int speed;
          aircraft description;
        } an_aircraft;
```

This example defines a base union `aircraft` which may either be `jet`, `helicopter`, or
`cargoplane`.

In the `an_aircraft` structure there is a `kind` member which indicates which structure is being held at the time.

8.3 Coercion or Type-Casting

C is one of the few languages to allow *coercion*, that is forcing one variable of one type to be another type. C allows this using the cast operator `()`. So:

```
int integernumber;
float floatnumber=9.87;

      integernumber=(int)floatnumber;
```

assigns 9 (the fractional part is thrown away) to `integernumber`.

And:

```
int integernumber=10;
float floatnumber;

      floatnumber=(float)integernumber;
```

assigns 10.0 to `floatnumber`.

Coercion can be used with any of the simple data types including `char`, so:

```
int integernumber;
char letter='A';

     integernumber=(int)letter;
```

assigns 65 (the ASCII code for 'A') to `integernumber`.

Some typecasting is done automatically — this is mainly with integer compatibility.

A good rule to follow is: **If in doubt cast**.

Another use is the make sure division behaves as requested: If we have two integers `internumber` and `anotherint` and we want the answer to be a float then :

e.g.
```
floatnumber =
 (float) internumber / (float) anotherint;
```

ensures floating point division.

8.4 Enumerated Types

Enumerated types contain a list of constants that can be addressed in integer values.

We can declare types and variables as follows.

```
enum days {mon, tues, ..., sun} week;
enum days week1, week2;
```

NOTE: As with arrays first enumerated name has index value 0. So `mon` has value 0, `tues` 1, and so on.

`week1` and `week2` are variables.

We can define other values:

```
enum escapes { bell = '\a',
               backspace = '\b', tab = '\t',
               newline = '\n', vtab = '\v',
               return = '\r'};
```

We can also override the 0 start value:

```
enum months {jan = 1, feb, mar, ......, dec};
```

Here it is implied that feb = 2 *etc.*

8.5 Static Variables

A **static** variable is local to particular function. However, it is only initialised once (on the first call to function).

Also the value of the variable on leaving the function remains **intact**. On the next call to the function the the `static` variable has the same value as on leaving.

To define a `static` variable simply prefix the variable declaration with the `static` keyword. For example:

```
void stat(); /* prototype fn */

main()
  { int i;

    for (i=0;i<5;++i)
        stat();
  }


stat()
  { int auto_var = 0;
    static int static_var = 0;

    printf( ``auto = %d, static = %d \n'',
            auto_var, static_var);
    ++auto_var;
    ++static_var;
  }
```

Output is:

```
auto_var = 0, static_var= 0
auto_var = 0, static_var = 1
auto_var = 0, static_var = 2
auto_var = 0, static_var = 3
auto_var = 0, static_var = 4
```

Clearly the `auto_var` variable is created each time. The `static_var` is created once and remembers its value.

8.6 Exercises

Exercise 8.1 *Write program using enumerated types which when given today's date will print out tomorrow's date in the for 31st January, for example.*

Exercise 8.2 *Write a simple database program that will store a persons details such as age, date of birth, address* etc.

Chapter 9

Pointers

Pointer are a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers.

C uses *pointers* a lot. **Why?**:

- It is the only way to express some computations.
- It produces compact and efficient code.
- It provides a very powerful tool.

C uses pointers explicitly with:

- Arrays,
- Structures,
- Functions.

NOTE: Pointers are perhaps the most difficult part of C to understand. C's implementation is slightly different DIFFERENT from other languages.

9.1 What is a Pointer?

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

The *unary* or *monadic* operator & gives the "address of a variable".

The *indirection* or dereference operator * gives the "contents of an object *pointed to* by a pointer".

To declare a pointer to a variable do:

```
int *pointer;
```

NOTE: We must associate a pointer to a particular type: You can't assign the address of a **short int** to a **long int**, for instance.

Consider the effect of the following code:

```
int x = 1, y = 2;
int *ip;

ip = &x;


y = *ip;


x = ip;


*ip = 3;
```

It is worth considering what is going on at the *machine level* in memory to fully understand how pointer work. Consider Fig. 9.1. Assume for the sake of this discussion that variable `x` resides at memory location 100, `y` at 200 and `ip` at 1000. **Note** A pointer is a variable and thus its values need to be stored somewhere. It is the nature of the pointers value that is *new.*

Now the assignments `x = 1` and `y = 2` obviously load these values into the variables. `ip` is declared to be a *pointer to an integer* and is assigned to the address of `x` (`&x`). So ip gets loaded with the value 100.

Next `y` gets assigned to the *contents of* `ip`. In this example `ip` currently *points* to memory location 100 — the location of `x`. So `y` gets assigned to the values of `x` — which is 1.

We have already seen that C is not too fussy about assigning values of different type. Thus it is perfectly **legal** (although not all that common) to assign the current value of `ip` to `x`. The value of `ip` at this instant is 200.

Finally we can assign a value to the contents of a pointer (*`ip`).

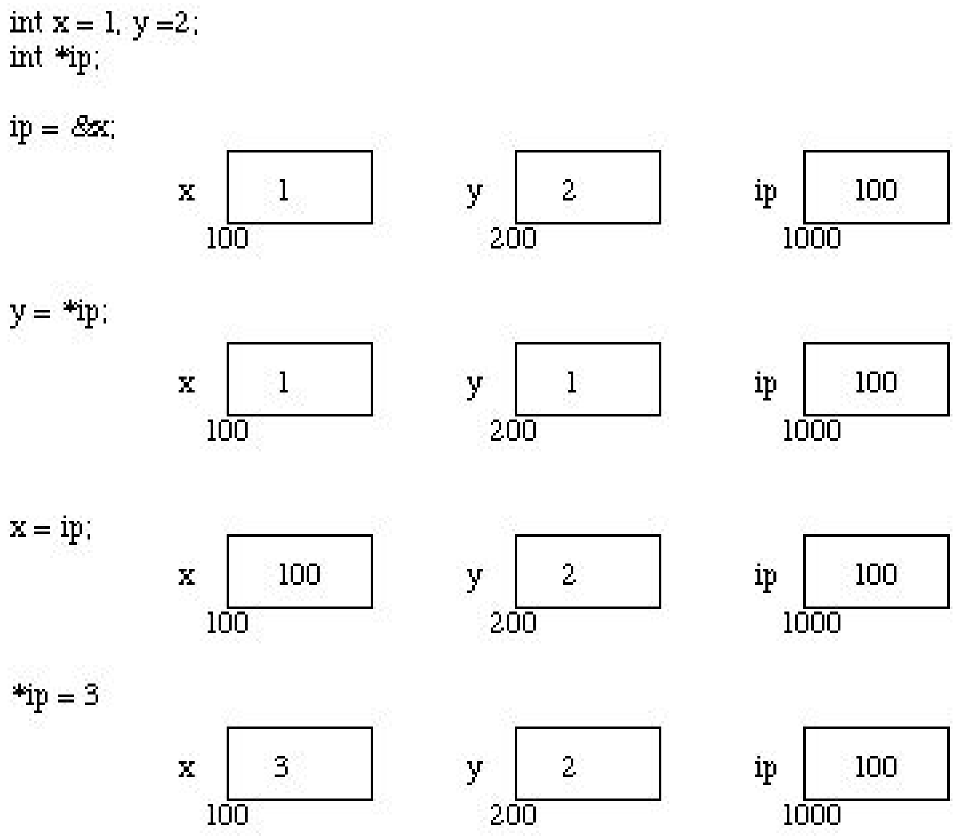


Figure 9.1: Pointer, Variables and Memory

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it.

So ...

```
int *ip;

*ip = 100;
```

will generate an error (program crash!!).

The correct use is:

```
int *ip;
int x;

ip = &x;
*ip = 100;
```

We can do integer arithmetic on a pointer:

```
float *flp, *flq;

*flp = *flp + 10;

++*flp;

(*flp)++;

flq = flp;
```

NOTE: A pointer to any variable type is an address in memory — which is an integer address. A pointer is definitely NOT an integer.

The reason we associate a pointer to a data type is so that it knows how many bytes the data is stored in. When we increment a pointer we increase the pointer by one "block" memory.

So for a character pointer ++ch_ptr adds 1 byte to the address.

For an integer or float ++ip or ++flp adds 4 bytes to the address.

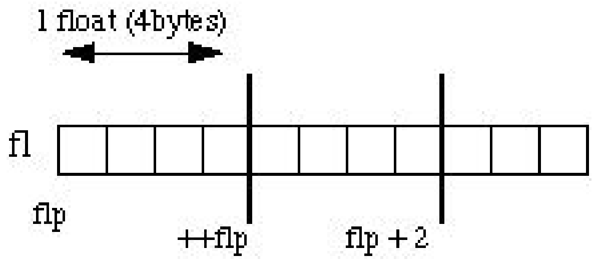


Figure 9.2: Pointer Arithmetic

Consider a float variable (`fl`) and a pointer to a float (`flp`) as shown in Fig. 9.2.

Assume that `flp` points to `fl` then if we increment the pointer (++`flp`) it moves to the position shown 4 bytes on. If on the other hand we added 2 to the pointer then it moves 2 **float positions** *i.e* 8 bytes as shown in the Figure.

9.2 Pointer and Functions

Let us now examine the close relationship between pointers and C's other major parts. We will start with functions.

When C passes arguments to functions it passes them by value.

There are many cases when we may want to alter a passed argument in the function and receive the new value back once to function has finished. Other languages do this (*e.g.* `var` parameters in PASCAL). C uses pointers explicitly to do this. Other languages mask the fact that pointers also underpin the implementation of this.

The best way to study this is to look at an example where we must be able to receive changed parameters.

Let us try and write a function to swap variables around?

The usual function *call*:

`swap(a, b)` WON'T WORK.

Pointers provide the solution: *Pass the address of the variables to the functions and access address of function.*

Thus our function call in our program would look like this:

```
   swap(&a, &b)
```

The Code to swap is fairly straightforward:

```
  void swap(int *px, int *py)

          { int temp;

            temp = *px;
            /* contents of pointer */

            *px = *py;
            *py = temp;
       }
```

We can return pointer from functions. A common example is when passing back structures. *e.g.*:

```
typedef struct {float x,y,z;} COORD;

main()

  {  COORD p1, *coord_fn();
     /* declare fn to return ptr of
     COORD type */

     ....
     p1 = *coord_fn(...);
  /* assign contents of address returned */
     ....
  }


   COORD *coord_fn(...)

     {  COORD p;

        .....
```

```
      p = ....;
      /* assign structure values */

      return &p;
      /* return address of p */
   }
```

Here we return a pointer whose contents are immediately *unwrapped* into a variable. We must do this straight away as the variable we pointed to was local to a function that has now finished. This means that the address space is free and can be overwritten. It will not have been overwritten straight after the function ha squit though so this is perfectly safe.

9.3 Pointers and Arrays

Pointers and arrays are very closely linked in C.

Hint: think of array elements arranged in consecutive memory locations.

Consider the following:

```
int a[10], x;
int *pa;

pa = &a[0]; /* pa pointer to address of a[0] */

x = *pa;
/* x = contents of pa (a[0] in this case) */
```

To get somewhere in the array (Fig. 9.3) using a pointer we could do:

`pa + i` ≡ `a[i]`

WARNING: There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.

C however is much more subtle in its link between arrays and pointers.

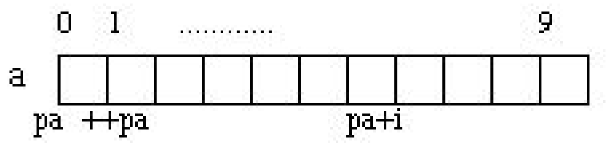


Figure 9.3: Arrays and Pointers

For example we can just type

`pa = a;`

instead of

`pa = &a[0]`

and

`a[i]` can be written as `*(a + i)`.
i.e. `&a[i]` ≡ `a + i`.

We also express pointer addressing like this:

`pa[i]` ≡ `*(pa + i)`.

However pointers and arrays are different:

- A pointer is a variable. We can do
  `pa = a` and `pa++`.

- An Array <u>is not</u> a variable. `a = pa` and `a++` ARE ILLEGAL.

This stuff is very important. Make sure you understand it. We will see a lot more of this.

We can now understand how arrays are passed to functions.

When an array is passed to a function what is actually passed is its initial elements location in memory.

So:

`strlen(s)` ≡ `strlen(&s[0])`

This is why we declare the function:

```
int strlen(char s[]);
```

An equivalent declaration is : `int strlen(char *s);`
since `char s[]` ≡ `char *s`.

`strlen()` is a *standard library* function (Chapter 18) that returns the length of a string. Let's look at how we may write a function:

```
int strlen(char *s)
   { char *p = s;

     while (*p != '\0');
        p++;
     return p-s;
   }
```

Now lets write a function to copy a string to another string. `strcpy()` is a standard library function that does this.

```
void strcpy(char *s, char *t)
   {  while ( (*s++ = *t++) != '\0);}
```

This uses pointers and assignment by value.

Very Neat!!

NOTE: Uses of Null statements with `while`.

9.4 Arrays of Pointers

We can have arrays of pointers since pointers are variables.

Example use:

Sort lines of text of different length.

NOTE: Text can't be moved or compared in a single operation.

Arrays of Pointers are a data representation that will cope efficiently and conveniently with variable length text lines.

How can we do this?:

- Store lines end-to-end in one big `char` array (Fig. 9.4). `\n` will delimit lines.
- Store pointers in a different array where each pointer points to 1st char of each new line.
- Compare two lines using `strcmp()` standard library function.
- If 2 lines are out of order — swap pointer in pointer array (<u>not text</u>).

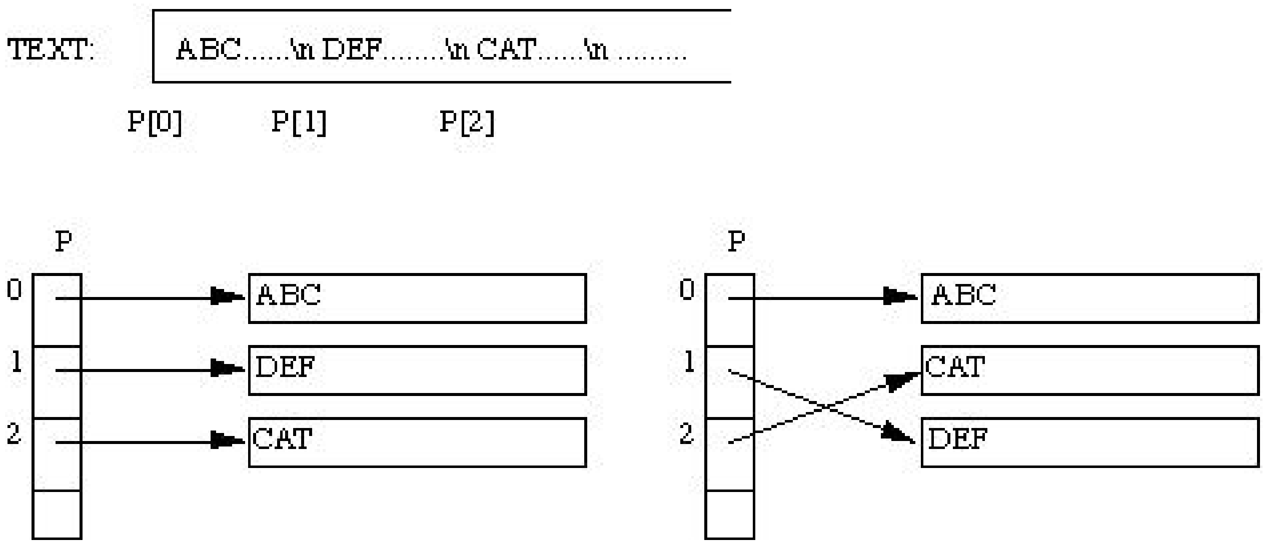


Figure 9.4: Arrays of Pointers (String Sorting Example)

This eliminates:

- complicated storage management.
- high overheads of moving lines.

9.5 Multidimensional arrays and pointers

We should think of multidimensional arrays in a different way in C:

A 2D array is really a 1D array, each of whose elements is itself an array

Hence

`a[n][m]` notation.

Array elements are stored row by row.

When we pass a 2D array to a function we must specify the number of columns — the number of rows is irrelevant.

The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.

Consider`int a[5][35]` to be passed in a function:

We can do:

```
f(int a[][35]) {.....}
```

or even:

```
f(int (*a)[35]) {.....}
```

We need parenthesis (*a) since [] have a higher precedence than *

So:

`int (*a)[35];` declares a pointer to an array of 35 `int`s.

`int *a[35];` declares an array of 35 pointers to `int`s.

Now lets look at the (subtle) difference between pointers and arrays. Strings are a common application of this.

Consider: `char *name[10];`

`char Aname[10][20];`

We can legally do `name[3][4]` and `Aname[3][4]` in C.
However

- `Aname` is a true 200 element 2D char array.
- access elements via
  $20{*}\mathit{row} + \mathit{col} + \mathit{base\_address}$
  in memory.
- `name` has 10 pointer elements.

NOTE: If each pointer in `name` is set to point to a 20 element array then and only then will 200 chars be set aside (+ 10 elements).

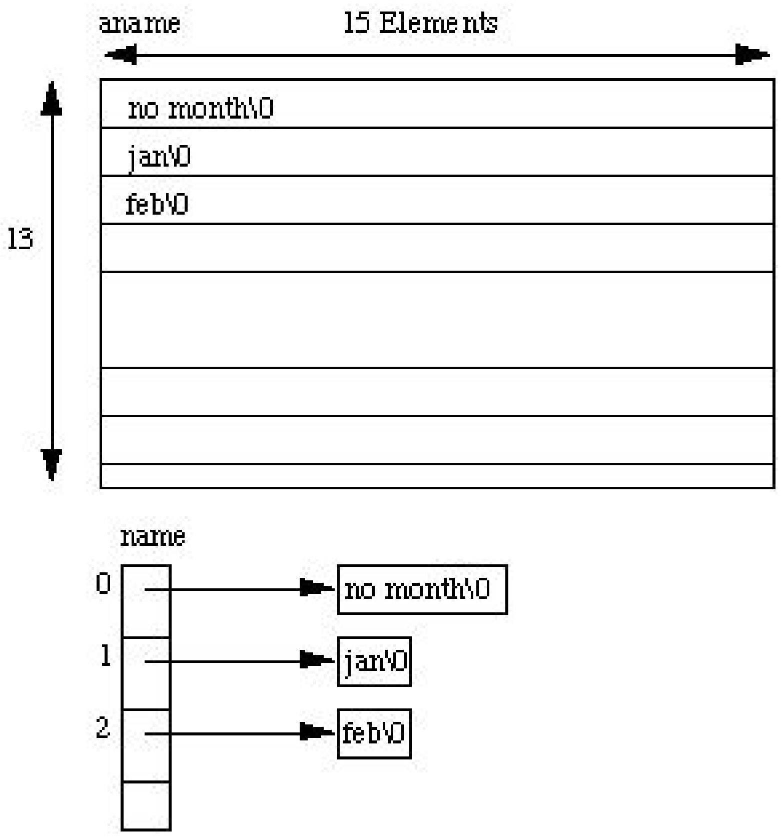


Figure 9.5: 2D Arrays and Arrays of Pointers

The advantage of the latter is that each pointer can point to arrays be of different length.

Consider:

```
char *name[] = { ‘‘no month’’, ‘‘jan’’,
                 ‘‘feb’’, ...  };
char Aname[][15] = { ‘‘no month’’, ‘‘jan’’,
                     ‘‘feb’’, ...  };
```

9.6 Static Initialisation of Pointer Arrays

Initialisation of arrays of pointers is an ideal application for an internal static array.

```
some_fn()
 { static char *months = { ``no month'',
                           ``jan'', ``feb'',
                           ...  };

 }
```

`static` reserves a private permanent bit of memory.

9.7 Pointers and Structures

These are fairly straight forward and are easily defined. Consider the following:

```
struct COORD {float x,y,z;} pt;
struct COORD *pt_ptr;


pt_ptr = &pt; /* assigns pointer to pt */
```

the −> operator lets us access a member of the structure pointed to by a pointer.*i.e.*:

```
    pt_ptr->x = 1.0;
    pt_ptr->y = pt_ptr->y - 3.0;
```

Example: Linked Lists

```
typedef struct {  int value;
                  ELEMENT *next;
               } ELEMENT;
```

```
ELEMENT n1, n2;


n1.next = &n2;
```

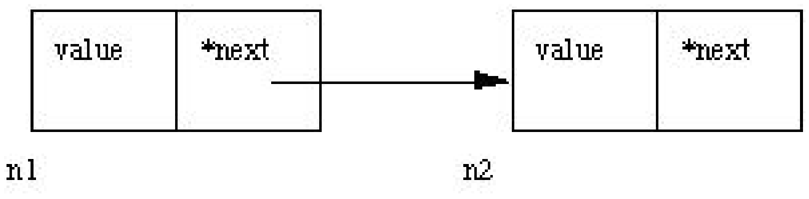


Figure 9.6: Linking Two Nodes

NOTE: We can only declare `next` as a pointer to `ELEMENT`. We cannot have a element of the variable type as this would set up a *recursive* definition which is **NOT ALLOWED**. We are allowed to set a pointer reference since 4 bytes are set aside for any pointer.

The above code links a node `n1` to `n2` (Fig. 9.6) we will look at this matter further in the next Chapter.

9.8 Common Pointer Pitfalls

Here we will highlight two common mistakes made with pointers.

9.8.1 Not assigning a pointer to memory address before using it

```
int *x;

*x = 100;
```

we need a physical location say: `int y;`

```
x = &y;
*x = 100;
```

This may be hard to spot. **NO COMPILER ERROR**. Also x could some random address at initialisation.

9.8.2 Illegal indirection

Suppose we have a function `malloc()` which tries to allocate memory dynamically (at run time) and returns a pointer to block of memory requested if successful or a `NULL` pointer otherwise.

`char *malloc()` — a standard library function (see later).

Let us have a pointer: `char *p`;

Consider:

```
*p = (char *) malloc(100);    /* request 100 bytes of memory */
*p = 'y';
```

There is mistake above. What is it?

No * in

$\underline{\texttt{*p}}$ `= (char *) malloc(100);`

Malloc returns a pointer. Also `p` does not point to any address.

The correct code should be:

```
p = (char *) malloc(100);
```

If code rectified one problem is if no memory is available and `p` is `NULL`. Therefore we can't do: `*p = 'y';`.

A good C program would check for this:

```
p = (char *) malloc(100);
if ( p == NULL)
  { printf(``Error: Out of Memory \n'');
    exit(1);
  }
*p = 'y';
```

9.9 Exercise

Exercise 9.1 *Write a C program to read through an array of any type using pointers. Write a C program to scan through this array to find a particular value.*

Exercise 9.2 *Write a program to find the number of times that a given word(i.e. a short string) occurs in a sentence (i.e. a long string!).*

Read data from standard input. The first line is a single word, which is followed by general text on the second line. Read both up to a newline character, and insert a terminating null before processing.

Typical output should be:

```
 The word is "the".
 The sentence is "the cat sat on the mat".
The word occurs 2 times.
```

Exercise 9.3 *Write a program that takes three variable (a, b, b) in as separate parameters and rotates the values stored so that value a goes to be, b, to c and c to a.*

Chapter 10

Dynamic Memory Allocation and Dynamic Structures

Dynamic allocation is a pretty unique feature to C (amongst high level languages). It enables us to create data types and structures of any size and length to suit our programs need within the program.

We will look at two common applications of this:

- dynamic arrays
- dynamic data structure *e.g.* linked lists

10.1 Malloc, Sizeof, and Free

The Function `malloc` is most commonly used to attempt to “grab” a continuous portion of memory. It is defined by:

```
void *malloc(size_t number_of_bytes)
```

That is to say it returns a pointer of type `void *` that is the start in memory of the reserved portion of size `number_of_bytes`. If memory cannot be allocated a `NULL` pointer is returned.

Since a `void *` is returned the C standard states that this pointer can be converted to any type. The `size_t` argument type is defined in `stdlib.h` and is an *unsigned type*.

So:

```
char *cp;
cp = malloc(100);
```

attempts to get 100 bytes and assigns the start address to `cp`.

Also it is usual to use the `sizeof()` function to specify the number of bytes:

```
int *ip;
ip = (int *) malloc(100*sizeof(int));
```

Some C compilers may require to cast the type of conversion. The `(int *)` means coercion to an integer pointer. Coercion to the correct pointer type is very important to ensure pointer arithmetic is performed correctly. I personally use it as a means of ensuring that I am totally correct in my coding and use cast all the time.

It is good practice to use `sizeof()` even if you know the actual size you want — it makes for device independent (portable) code.

`sizeof` can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function.

SO:

```
int i;
struct COORD {float x,y,z};
typedef struct COORD PT;

sizeof(int), sizeof(i),
sizeof(struct COORD) and
sizeof(PT) are all ACCEPTABLE
```

In the above we can use the link between pointers and arrays to treat the reserved memory like an array. *i.e* we can do things like:

```
ip[0] = 100;
```

or

```
    for(i=0;i<100;++i) scanf("%d",ip++);
```

When you have finished using a portion of memory you should always `free()` it. This allows the memory *freed* to be aavailable again, possibly for further `malloc()` calls

The function `free()` takes a pointer as an argument and frees the memory to which the pointer refers.

10.2 Calloc and Realloc

There are two additional memory allocation functions, `Calloc()` and `Realloc()`. Their prototypes are given below:

```
void *calloc(size_t num_elements, size_t element_size};

void *realloc( void *ptr, size_t new_size);
```

`Malloc` does not initialise memory (to *zero*) in any way. If you wish to initialise memory then use `calloc`. Calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between `calloc` and `malloc` in that `calloc` takes the number of desired elements, `num_elements`, and element_size, `element_size`, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would do:

```
    int *ip;
    ip = (int *) calloc(100, sizeof(int));
```

`Realloc` is a function which attempts to change the size of a previous allocated block of memory. The new size can be larger or smaller. If the block is made larger then the old contents remain unchanged and memory is added to the end of the block. If the size is made smaller then the remaining contents are unchanged.

If the original block size cannot be resized then `realloc` will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You **must** use this new value. If new memory cannot be reallocated then `realloc` returns `NULL`.

Thus to change the size of memory allocated to the `*ip` pointer above to an array block of 50 integers instead of 100, simply do:

```
ip = (int *) calloc( ip, 50);
```

10.3 Linked Lists

Let us now return to our linked list example:

```
typedef struct {  int value;
                  ELEMENT *next;
               } ELEMENT;
```

We can now try to grow the list dynamically:

```
link = (ELEMENT *) malloc(sizeof(ELEMENT));
```

This will allocate memory for a new `link`.

If we want to deassign memory from a pointer use the `free()` function:

```
free(link)
```

See Example programs (`queue.c`*) below and try exercises for further practice.*

10.4 Full Program: `queue.c`

A queue is basically a special case of a linked list where one data element joins the list at the left end and leaves in a ordered fashion at the other end.

The full listing for `queue.c` is as follows:

```
/*      */
/* queue.c       */
/* Demo of dynamic data structures in C                      */
```

```
#include <stdio.h>

#define FALSE 0
#define NULL 0

typedef struct {
    int     dataitem;
    struct listelement *link;
}               listelement;

void Menu (int *choice);
listelement * AddItem (listelement * listpointer, int data);
listelement * RemoveItem (listelement * listpointer);
void PrintQueue (listelement * listpointer);
void ClearQueue (listelement * listpointer);

main () {
    listelement listmember, *listpointer;
    int     data,
            choice;

    listpointer = NULL;
    do {
Menu (&choice);
switch (choice) {
    case 1:
printf ("Enter data item value to add  ");
scanf ("%d", &data);
listpointer = AddItem (listpointer, data);
break;
    case 2:
if (listpointer == NULL)
    printf ("Queue empty!\n");
else
    listpointer = RemoveItem (listpointer);
break;
    case 3:
PrintQueue (listpointer);
break;

    case 4:
break;

    default:
printf ("Invalid menu choice - try again\n");
break;
}
    } while (choice != 4);
    ClearQueue (listpointer);
} /* main */

void Menu (int *choice) {

    char    local;

    printf ("\nEnter\t1 to add item,\n\t2 to remove item\n\
\t3 to print queue\n\t4 to quit\n");
    do {
local = getchar ();
if ((isdigit (local) == FALSE) && (local != '\n')) {
    printf ("\nyou must enter an integer.\n");
    printf ("Enter 1 to add, 2 to remove, 3 to print, 4 to quit\n");
}
    } while (isdigit ((unsigned char) local) == FALSE);
    *choice = (int) local - '0';
}

listelement * AddItem (listelement * listpointer, int data) {

    listelement * lp = listpointer;

    if (listpointer != NULL) {
while (listpointer -> link != NULL)
    listpointer = listpointer -> link;
listpointer -> link = (struct listelement  *) malloc (sizeof (listelement))
listpointer = listpointer -> link;
listpointer -> link = NULL;
listpointer -> dataitem = data;
return lp;
    }
    else {
listpointer = (struct listelement  *) malloc (sizeof (listelement));
listpointer -> link = NULL;
listpointer -> dataitem = data;
return listpointer;
    }
}

listelement * RemoveItem (listelement * listpointer) {

    listelement * tempp;
    printf ("Element removed is %d\n", listpointer -> dataitem);
    tempp = listpointer -> link;
    free (listpointer);
    return tempp;
}

void PrintQueue (listelement * listpointer) {

    if (listpointer == NULL)
printf ("queue is empty!\n");
    else
while (listpointer != NULL) {
    printf ("%d\t", listpointer -> dataitem);
    listpointer = listpointer -> link;
}
    printf ("\n");
}

void ClearQueue (listelement * listpointer) {

    while (listpointer != NULL) {
listpointer = RemoveItem (listpointer);
    }
```

```
}
```

10.5 Exercises

Exercise 10.1 *Write a program that reads a number that says how many integer numbers are to be stored in an array, creates an array to fit the exact size of the data and then reads in that many numbers into the array.*

Exercise 10.2 *Write a program to implement the linked list as described in the notes above.*

Exercise 10.3 *Write a program to sort a sequence of numbers using a binary tree (Using Pointers). A binary tree is a tree structure with only two (possible) branches from each node (Fig. 10.1). Each branch then represents a false or true decision. To sort numbers simply assign the left branch to take numbers less than the node number and the right branch any other number (greater than or equal to). To obtain a sorted list simply search the tree in a depth first fashion.*

Your program should: Create a binary tree structure. Create routines for loading the tree appropriately. Read in integer numbers terminated by a zero. Sort numbers into numeric ascending order. Print out the resulting ordered values, printing ten numbers per line as far as possible.

Typical output should be

```
The sorted values are:
 2  4  6  6  7  9 10 11 11 11
15 16 17 18 20 20 21 21 23 24
27 28 29 30
```

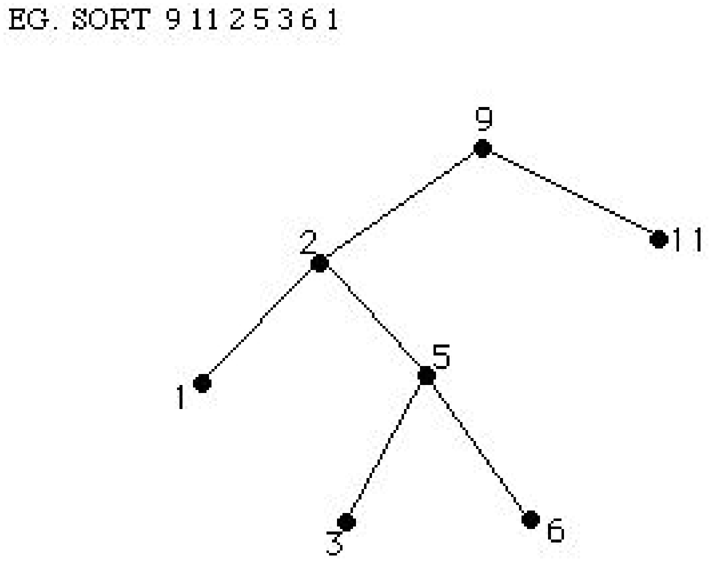


Figure 10.1: Example of a binary tree sort

Chapter 11

Advanced Pointer Topics

We have introduced many applications and techniques that use pointers. We have introduced some advanced pointer issues already. This chapter brings together some topics we have briefly mentioned and others to complete our study C pointers.

In this chapter we will:

- Examine pointers to pointers in more detail.
- See how pointers are used in command line input in C.
- Study pointers to functions

11.1 Pointers to Pointers

We introduced the concept of a pointer to a pointer previously. You can have a pointer to a pointer of any type.

Consider the following:

```
char ch;  /* a character */
char *pch; /* a pointer to a character */
char **ppch; /* a pointer to a pointer to a character */
```

We can visualise this in Figure 11.1. Here we can see that `**ppch` refers to memory address of `*pch` which refers to the memory address of the variable `ch`. But what does this mean in practice?

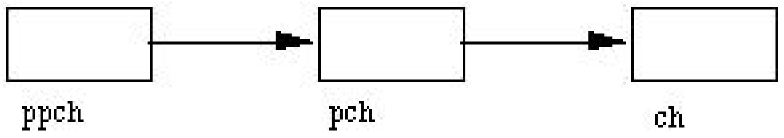


Figure 11.1: Pointers to pointers

Recall that `char *` refers to a (`NULL` terminated string. So one common and convenient notion is to declare a pointer to a pointer to a string (Figure 11.2)

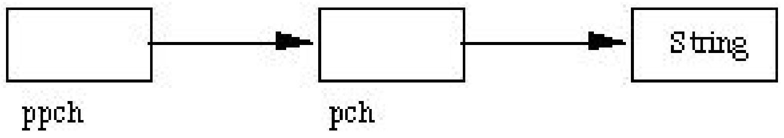


Figure 11.2: Pointer to String

Taking this one stage further we can have several strings being pointed to by the pointer (Figure 11.3)

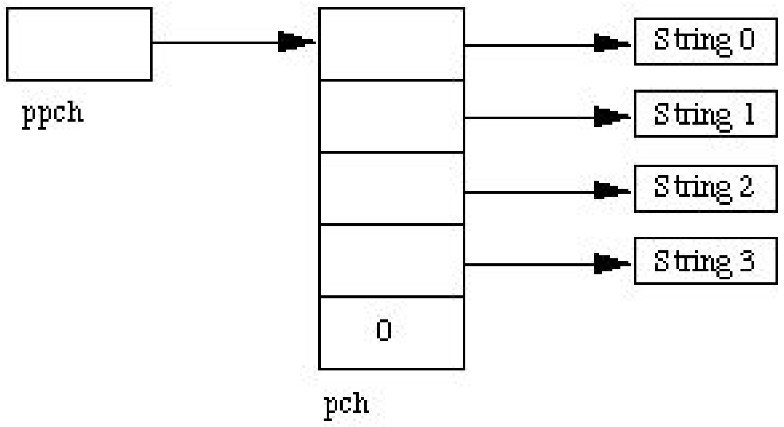


Figure 11.3: Pointer to Several Strings

We can refer to individual strings by `ppch[0]`, `ppch[1]`, ... Thus this is identical to declaring `char *ppch[]`.

One common occurrence of this type is in C command line argument input which we now consider.

11.2 Command line input

C lets read arguments from the command line which can then be used in our programs.

We can type arguments after the program name when we run the program.

We have seen this with the compiler for example

```
c89 -o prog prog.c
```

`c89` is the program, `-o prog prog.c` the arguments.

In order to be able to use such arguments in our code we must define them as follows:

```
main(int argc, char **argv)
```

So our `main` function now has its own arguments. These are the only arguments main accepts.

- **argc** is the number of arguments typed — including the program name.
- **argv** is an array of strings holding each command line argument — including the program name in the first array element.

A simple program example:

```
#include<stdio.h>

main (int argc, char **argv)
    { /* program to print arguments
      from command line */

      int i;

      printf(``argc = %d\n\n'',argc);
      for (i=0;i<argc;++i)
          printf(``argv[%d]:  %s\n'',
                 i, argv[i]);
    }
```

Assume it is compiled to run it as args.

So if we type:

```
args f1 ``f2'' f3 4 stop!
```

The output would be:

```
argc = 6

argv[0] = args
argv[1] = f1
argv[2] = f2
argv[3] = f3
argv[4] = 4
argv[5] = stop!
```

NOTE: • argv[0] is program name.
• argc counts program name
• Embedded " " are ignored.
Blank spaces delimit end of arguments.
Put blanks in " " if needed.

11.3 Pointers to a Function

Pointer to a function are perhaps on of the more confusing uses of pointers in C. Pointers to functions are not as common as other pointer uses. However, one common use is in a passing pointers to a function as a parameter in a function call. (Yes this is getting confusing, hold on to your hats for a moment).

This is especially useful when alternative functions maybe used to perform similar tasks on data. You can pass the data and the function to be used to some *control* function for instance. As we will see shortly the C standard library provided some basic sorting (`qsort`) and searching (`bsearch`) functions for free. You can easily embed your own functions.

To declare a pointer to a function do:

```
int (*pf) ();
```

This simply declares a pointer `*pf` to function that returns and `int`. No actual function is *pointed* to yet.

If we have a function `int f()` then we may simply (!!) write:

```
pf = &f;
```

For compiler prototyping to fully work it is better to have full function prototypes for the function and the pointer to a function:

```
int f(int);
int (*pf) (int) = &f;
```

Now `f()` returns an `int` and takes one `int` as a parameter.

You can do things like:

```
ans = f(5);
ans = pf(5);
```

which are equivalent.

The `qsort` standard library function is very useful function that is designed to sort an array by a *key* value of *any type* into ascending order, as long as the elements of the array are of fixed type.

qsort is prototyped in (`stdlib.h`):

```
void qsort(void *base, size_t num_elements, size_t element_size,
   int (*compare)(void const *, void  const *));
```

The argument `base` points to the array to be sorted, `num_elements` indicates how long the array is, `element_size` is the size in bytes of each array element and the final argument `compare` is a pointer to a function.

`qsort` calls the `compare` function which is user defined to compare the data when sorting. Note that `qsort` maintains it's data type independence by giving the comparison responsibility to the user. The compare function must return certain (`integer`) values according to the comparison result:

less than zero : if first value is less than the second value

zero : if first value is equal to the second value

greater than zero : if first value is greater than the second value

Some quite complicated data structures can be sorted in this manner. For example, to sort the following structure by `integer` key:

```
typedef struct {
        int   key;
struct other_data;
} Record;
```

We can write a compare function, `record_compare`:

```
int record\_compare(void const *a, void  const *a)
  {  return ( ((Record *)a)->key - ((Record *)b)->key );
  }
```

Assuming that we have an `array` of `array_length Records` suitably filled with date we can call `qsort` like this:

```
qsort( array, arraylength, sizeof(Record), record_compare);
```

Further examples of standard library and system calls that use pointers to functions may be found in Chapters 15.4 and 19.1.

11.4 Exercises

Exercise 11.1 *Write a program last that prints the last n lines of its text input. By default n should be 5, but your program should allow an optional argument so that*

```
last -n
```

prints out the last n lines, where n is any integer. Your program should make the best use of available storage. (Input of text could be by reading a file specified from the command or reading a file from standard input)

Exercise 11.2 *Write a program that sorts a list of integers in ascending order. However if a -r flag is present on the command line your program should sort the list in descending order. (You may use any sorting routine you wish)*

Exercise 11.3 *Write a program that reads the following structure and sorts the data by keyword using* `qsort`

```
typedef struct {
        char   keyword[10];
int    other_data;
} Record;
```

Exercise 11.4 *An* insertion sort *is performed by adding values to an array one by one. The first value is simply stored at the beginning of the array. Each subsequent value is added by finding its ordered position in the array, moving data as needed to accommodate the value and inserting the value in this position.*

Write a function called `insort` *that performs this task and behaves in the same manner as* `qsort`*,* i.e *it can sort an array by a* key *value of* any type *and it has similar prototyping.*

Chapter 12

Low Level Operators and Bit Fields

We have seen how pointers give us control over low level memory operations.

Many programs (*e.g.* systems type applications) must actually operate at a low level where individual bytes must be operated on.

NOTE: The combination of pointers and bit-level operators makes C useful for many low level applications and can almost replace assembly code. (Only about 10 % of UNIX is assembly code the rest is C!!.)

12.1 Bitwise Operators

The *bitwise* operators of C a summarised in the following table:

&	AND
|	OR
$\wedge$	XOR
$\sim$	One's Compliment $0 \rightarrow 1$ $1 \rightarrow 0$
<<	Left shift
>>	Right Shift

Table 12.1: Bitwise operators

DO NOT confuse & with &&: & is bitwise AND, && logical AND. Similarly for | and ||.

∼ is a unary operator — it only operates on one argument to right of the operator.

The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator must be positive. The vacated bits are filled with zero (*i.e.* There is **NO** wrap around).

For example: $x << 2$ shifts the bits in x by 2 places to the left.

So:

if $x = 00000010$ (binary) or 2 (decimal)

then:

$x >>= 2 \Rightarrow x = 00000000$ or 0 (decimal)

Also: if $x = 00000010$ (binary) or 2 (decimal)

$x <<= 2 \Rightarrow x = 00001000$ or 8 (decimal)

Therefore a shift left is equivalent to a multiplication by 2.

Similarly a shift right is equal to division by 2

NOTE: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 *use shifts.*

To illustrate many points of bitwise operators let us write a function, `Bitcount`, that counts bits set to 1 in an 8 bit number (`unsigned char`) passed as an argument to the function.

```
int bitcount(unsigned char x)

   { int count;

     for (count=0; x != 0; x>>= 1);
         if ( x & 01)
            count++;
     return count;
   }
```

This function illustrates many C program points:

- `for` loop not used for simple counting operation
- `x>>= 1` ⇒ `x = x >> 1`
- for loop will repeatedly shift right `x` until `x` becomes 0
- use expression evaluation of `x & 01` to control `if`
- `x & 01` *masks* of 1st bit of `x` if this is 1 then `count++`

12.2 Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. *e.g.* 1 bit flags can be compacted — Symbol tables in compilers.
- Reading external file formats — non-standard file formats could be read in. *E.g.* 9 bit integers.

C lets us do this in a structure definition by putting :*bit length* after the variable. *i.e.*

```
struct packed_struct {
       unsigned int f1:1;
       unsigned int f2:1;
       unsigned int f3:1;
       unsigned int f4:1;
       unsigned int type:4;
       unsigned int funny_int:9;
} pack;
```

Here the `packed_struct` contains 6 members: Four 1 bit *flags* `f1..f3`, a 4 bit `type` and a 9 bit `funny_int`.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word (see comments on bit fiels portability below).

Access members as usual via:

```
pack.type = 7;
```

NOTE:

- Only n lower bits will be assigned to an n bit number. So type cannot take values larger than 15 (4 bits long).
- Bit fields are always converted to integer type for computation.
- You are allowed to mix "normal" types with bit fields.
- The `unsigned` definition is important - ensures that no bits are used as a $\pm$ flag.

12.2.1 Bit Fields: Practical Example

Frequently device controllers (*e.g.* disk drives) and the operating system need to communicate at a low level. Device controllers contain several *registers* which may be packed together in one integer (Figure 12.1).

We could define this register easily with bit fields:

```
struct DISK_REGISTER  {
     unsigned ready:1;
     unsigned error_occured:1;
     unsigned disk_spinning:1;
     unsigned write_protect:1;
     unsigned head_loaded:1;
     unsigned error_code:8;
     unsigned track:9;
     unsigned sector:5;
     unsigned command:5;
};
```

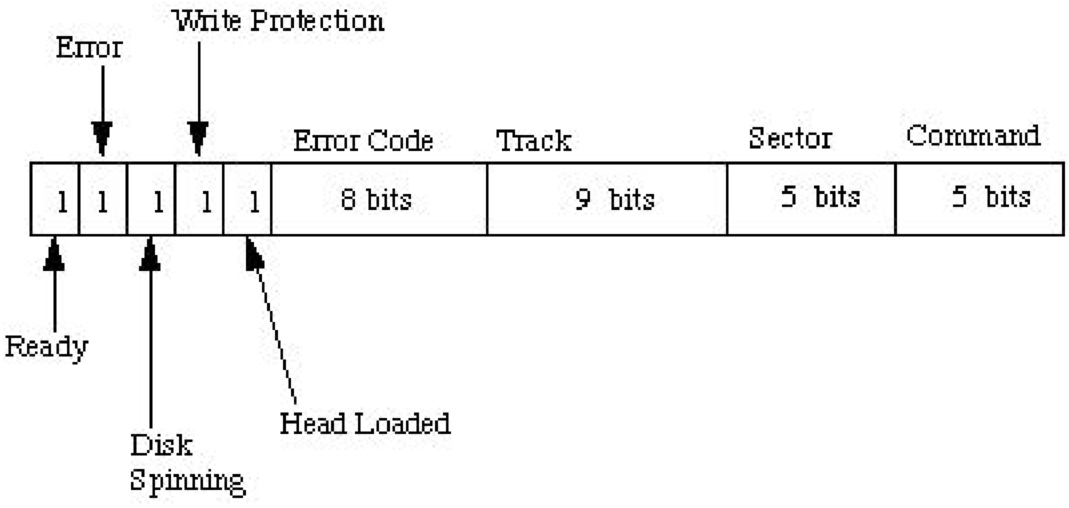


Figure 12.1: Example Disk Controller Register

To access values stored at a particular memory address, `DISK_REGISTER_MEMORY` we can assign a pointer of the above structure to access the memory via:

```
struct DISK_REGISTER *disk_reg = (struct DISK_REGISTER *) DISK_REGISTER_MEMORY;
```

The disk driver code to access this is now relatively straightforward:

```
/* Define sector and track to start read */

disk_reg->sector = new_sector;
disk_reg->track = new_track;
disk_reg->command = READ;

/* wait until operation done, ready will be true */

while ( ! disk_reg->ready ) ;

/* check for errors */

if (disk_reg->error_occured)
  { /* interrogate disk_reg->error_code for error type */
    switch (disk_reg->error_code)
    ......
  }
```

12.2.2 A note of caution: Portability

Bit fields are a convenient way to express many difficult operations. However, bit fields do suffer from a lack of portability between platforms:

- integers may be signed or unsigned
- Many compilers limit the maximum number of bits in the bit field to the size of an `integer` which may be either 16-bit or 32-bit varieties.
- Some bit field members are stored left to right others are stored right to left in memory.
- If bit fields too large, next bit field may be stored consecutively in memory (overlapping the boundary between memory locations) or in the next word of memory.

If portability of code is a premium you can use bit shifting and masking to achieve the same results but not as easy to express or read. For example:

```
unsigned int  *disk_reg = (unsigned int *) DISK_REGISTER_MEMORY;

/* see if disk error occured */

disk_error_occured = (disk_reg & 0x40000000) >> 31;
```

12.3 Exercises

Exercise 12.1 *Write a function that prints out an 8-bit (unsigned char) number in binary format.*

Exercise 12.2 *Write a function setbits(x,p,n,y) that returns x with the n bits that begin at position p set to the rightmost n bits of an unsigned char variable y (leaving other bits unchanged).*

E.g. if $x = 10101010$ *(170 decimal) and* $y = 10100111$ *(167 decimal) and* $n = 3$ *and* $p = 6$ *say then you need to strip off 3 bits of y (111) and put them in x at position* $10xxx010$ *to get answer* 10111010*.*

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
x = 10101010 (binary)
y = 10100111 (binary)
setbits n = 3, p = 6 gives x = 10111010 (binary)
```

Exercise 12.3 *Write a function that inverts the bits of an unsigned char x and stores answer in y.*

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
x = 10101010 (binary)
x inverted = 01010101 (binary)
```

Exercise 12.4 *Write a function that rotates (***NOT shifts***) to the right by n bit positions the bits of an unsigned char x.ie no bits are lost in this process.*

Your answer should print out the result in binary form (see Exercise 12.1 although input can be in decimal form.

Your output should be like this:

```
x = 10100111 (binary)
x rotated by 3 = 11110100 (binary)
```

Note: All the functions developed should be as concise as possible

Chapter 13

The C Preprocessor

Recall that preprocessing is the first step in the C program compilation stage — this feature is unique to C compilers.

The preprocessor more or less provides its own language which can be a very powerful tool to the programmer. Recall that all preprocessor directives or commands begin with a #.

Use of the preprocessor is advantageous since it makes:

- programs easier to develop,
- easier to read,
- easier to modify
- C code more transportable between different machine architectures.

The preprocessor also lets us customise the language. For example to replace { ... } block statements delimiters by PASCAL like `begin ... end` we can do:

```
#define begin {
#define end }
```

During compilation all occurrences of `begin` and `end` get replaced by corresponding { or } and so the subsequent C compilation stage does not know any difference!!!.

Lets look at `#define` in more detail

13.1 #define

Use this to define constants or any macro substitution. Use as follows:

```
#define <macro> <replacement name>
```

For Example

```
#define FALSE 0
#define TRUE !FALSE
```

We can also define small "functions" using `#define`. For example max. of two variables:

```
#define max(A,B) ( (A) > (B) ? (A):(B))
```

? is the ternary operator in C.

Note: that this does <u>not</u> define a proper function `max`.

All it means that wherever we place `max(C†,D†)` the text gets replaced by the appropriate definition. [† = any variable names – not necessarily C and D]

So if in our C code we typed something like:

```
x = max(q+r,s+t);
```

after preprocessing, if we were able to look at the code it would appear like this:

```
x = ( (q+r) > (r+s) ?  (q+r) :  (s+t));
```

Other examples of `#define` could be:

```
#define Deg_to_Rad(X) (X*M_PI/180.0)
/* converts degrees to radians, M_PI is the value
of pi and is defined in math.h library */

#define LEFT_SHIFT_8 <<8
```

NOTE: The last macro `LEFT_SHIFT_8` is only valid so long as replacement context is valid *i.e.* `x = y LEFT_SHIFT_8`.

13.2 #undef

This commands undefined a macro. A macro **must** be undefined before being redefined to a different value.

13.3 #include

This directive includes a file into code.

It has two possible forms:

```
    #include <file>
or
    #include ``file''
```

`<file>` tells the compiler to look where system include files are held. Usually UNIX systems store files in \usr\include\\ directory.

`` ``file'' `` looks for a file in the current directory (where program was run from)

`Include`*d* files usually contain C prototypes and declarations from header files and not (algorithmic) C code (SEE next Chapter for reasons)

13.4 #if — Conditional inclusion

`#if` evaluates a constant integer expression. You always need a `#endif` to delimit end of statement.

We can have *else etc.* as well by using `#else` and `#elif` — else if.

Another common use of `#if` is with:

`#ifdef` — if defined and

`#ifndef` — if not defined

These are useful for checking if macros are set — perhaps from different program modules and header files.

For example, to set integer size for a portable C program between TurboC (on MSDOS) and Unix (or other) Operating systems. Recall that TurboC uses 16 bits/integer and UNIX 32 bits/integer.

Assume that if TurboC is running a macro `TURBOC` will be defined. So we just need to check for this:

```
#ifdef TURBOC
  #define INT_SIZE 16
#else
  #define INT_SIZE 32
#endif
```

As another example if running program on MSDOS machine we want to include file msdos.h otherwise a default.h file. A macro `SYSTEM` is set (by OS) to type of system so check for this:

```
#if SYSTEM == MSDOS
  #include <msdos.h>
#else
  #include ``default.h''
#endif
```

13.5 Preprocessor Compiler Control

You can use the `cc` compiler to control what values are set or defined from the command line. This gives some flexibility in setting customised values and has some other useful functions. The `-D` compiler option is used. For example:

```
     cc -DLINELENGTH=80 prog.c -o prog
```

has the same effect as:

```
#define LINELENGTH 80
```

Note that any `#define` or `#undef` **within** the program (`prog.c` above) **override** command line settings.

You can also set a symbol without a value, for example:

```
cc -DDEBUG prog.c -o prog
```

Here the value is assumed to be 1.

The setting of such flags is useful, especially for debugging. You can put commands like:

```
#ifdef DEBUG
     print("Debugging: Program Version 1\");
#else
     print("Program Version 1 (Production)\");
#endif
```

Also since preprocessor command can be written anywhere in a C program you can filter out variables etc for printing *etc.* when debugging:

```
x = y *3;

#ifdef DEBUG
     print("Debugging: Variables (x,y) = \",x,y);
#endif
```

The `-E` command line is worth mentioning just for academic reasons. It is not that practical a command. The `-E command` will force the compiler to stop after the preprocessing stage and output the current state of your program. Apart from being debugging aid for preprocessor commands and also as a useful initial learning tool (try this option out with some of the examples above) it is not that commonly used.

13.6 Other Preprocessor Commands

There are few other preprocessor directives available:

`#error text of error message` — generates an appropriate compiler error message. *e.g*

```
#ifdef OS_MSDOS
  #include <msdos.h>
#elifdef OS_UNIX
  #include ``default.h''
#else
  #error Wrong OS!!
#endif
```

# **line** `number "string"` — informs the preprocessor that the `number` is the next number of line of input. `"string"` is optional and names the next line of input. This is most often used with programs that translate other languages to C. For example, error messages produced by the C compiler can reference the file name and line numbers of the original source files instead of the intermediate C (translated) source files.

13.7 Exercises

Exercise 13.1 *Define a preprocessor macro* `swap(t, x, y)` *that will swap two arguments* `x` *and* `y` *of a given type* `t`*.*

Exercise 13.2 *Define a preprocessor macro to select:*

- *the least significant bit from an* `unsigned char`
- *the nth (assuming least significant is 0) bit from an* `unsigned char`*.*

Chapter 14

C, UNIX and Standard Libraries

There is a very close link between C and most operating systems that run our C programs. Almost the whole of the UNIX operating system is written in C. This Chapter will look at how C and UNIX interface together. [1]

We have to use UNIX to maintain our file space, edit, compile and run programs *etc.*.

However UNIX is much more useful than this:

14.1 Advantages of using UNIX with C

- **Portability** — UNIX, or a variety of UNIX, is available on many machines. Programs written in *standard* UNIX and C should run on any of them with little difficulty.

- **Multiuser / Multitasking** — many programs can share a machines processing power.

- **File handling** — hierarchical file system with many file handling routines.

[1]Even though we deal with UNIX and C nearly all the forthcoming discussions are applicable to MSDOS and other operating systems

- **Shell Programming** — UNIX provides a powerful command interpreter that
  understands over 200 commands and can also run UNIX and user-defined programs.
- **Pipe** — where the output of one program can be made the input of another. This can done from command line or within a C program.
- **UNIX utilities** — there over 200 utilities that let you accomplish many routines without writing new programs. *e.g.* make, grep, diff, awk, more ....
- **System calls** — UNIX has about 60 system calls that are at the *heart* of the operating system or the *kernel* of UNIX. The calls are actually written in C. All of them can be accessed from C programs. Basic I/0, system clock access are examples. The function `open()` is an example of a system call.
- **Library functions** — additions to the operating system.

14.2 Using UNIX System Calls and Library Functions

To use system calls and library functions in a C program we simply call the appropriate C function.

Examples of standard library functions we have met include the higher level I/O functions — `fprintf()`, `malloc()` ...

Aritmetic operators, random number generators — `random(), srandom(), lrand48(), drand48()` *`etc.`* and basic C types to string conversion are memebers of the `stdlib.h` standard library.

All math functions such as `sin(), cos(), sqrt()` are standard math library (`math.h`) functions and others follow in a similar fashion.

For most system calls and library functions we have to include an appropriate header file. *e.g.* `stdio.h, math.h`

To use a function, ensure that you have made the required `#includes` in your C file. Then the function can be called as though you had defined it yourself.

It is important to ensure that your arguments have the expected types, otherwise the function will probably produce strange results. `lint` is quite good at checking such things.

Some libraries require extra options before the compiler can support their use. For example, to compile a program including functions from the `math.h` library the command might be

```
cc mathprog.c -o mathprog -lm
```

The final `-lm` is an instruction to link the maths library with the program. The manual page for each function will usually inform you if any special compiler flags are required.

Information on nearly all system calls and library functions is available in manual pages. These are available on line: Simply type `man` function name.

e.g. `man drand48`

would give information about this random number generator.

Over the coming chapters we will be investigating in detail many aspects of the C Standard Library and also other UNIX libraries.

Chapter 15

Integer Functions, Random Number, String Conversion, Searching and Sorting: `<stdlib.h>`

To use all functions in this library you must:

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

There are three basic categories of functions:

- Arithmetic
- Random Numbers
- String Conversion

The use of all the functions is relatively straightforward. We only consider them briefly in turn in this Chapter.

15.1 Arithmetic Functions

There are 4 basic integer functions:

```
int abs(int number);
long int labs(long int number);
```

```
div_t div(int numerator,int denominator);
ldiv_t ldiv(long int numerator, long int denominator);
```

Essentially there are two functions with integer and long integer compatibility.

`abs` functions return the absolute value of its `number` arguments. For example, abs(2) returns 2 as does abs(-2).

`div` takes two arguments, `numerator` and `denominator` and produces a quotient and a remainder of the integer division. The `div_t` structure is defined (in `stdlib.h`) as follows:

```
typedef struct {
        int  quot; /* quotient */
        int  rem;  /* remainder */
} div_t;
```

(`ldiv_t` is similarly defined).

Thus:

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

int num = 8, den = 3;
div_t ans;


ans = div(num,den);

printf("Answer:\n\t Quotient = %d\n\t Remainder = %d\n", \
ans.quot,ans.rem);
```

Produces the following output:

```
Answer:
Quotient = 2
  Remainder = 2
```

15.2 Random Numbers

Random numbers are useful in programs that need to simulate random events, such as games, simulations and experimentations. In practice no functions produce truly random data — they produce *pseudo-random* numbers. These are computed form a given formula (different generators use different formulae) and the number sequences they produce are repeatable. A *seed* is usually set from which the sequence is generated. Therefore is you set the same seed all the time the same set will be be computed.

One common technique to introduce further randomness into a random number generator is to use the time of the day to set the seed, as this will always be changing. (We will study the standard library time functions later in Chapter 20).

There are many (pseudo) random number functions in the standard library. They all operate on the same basic idea but generate different number sequences (based on different generator functions) over different number ranges.

The simplest set of functions is:

```
int rand(void);
void srand(unsigned int seed);
```

`rand()` returns successive pseudo-random numbers in the range from 0 to (2^{15})-1.

`srand()` is used to set the seed. A simple example of using the time of the day to initiate a seed is via the call:

```
srand( (unsigned int) time( NULL ));
```

The following program `card.c` illustrates the use of these functions to simulate a pack of cards being shuffled:

```
/*
** Use random numbers to shuffle the "cards" in the deck.  The second
** argument indicates the number of cards.  The first time this
** function is called, srand is called to initialize the random
** number generator.
*/
#include <stdlib.h>
```

```
#include <time.h>
#define TRUE 1
#define FALSE 0

void shuffle( int *deck, int n_cards )
{
int i;
static int first_time = TRUE;

/*
** Seed the random number generator with the current time
** of day if we haven't done so yet.
*/
if( first_time ){
first_time = FALSE;
srand( (unsigned int)time( NULL ) );
}

/*
** "Shuffle" by interchanging random pairs of cards.
*/
for( i = n_cards - 1; i > 0; i -= 1 ){
int where;
int temp;

where = rand() % i;
temp = deck[ where ];
deck[ where ] = deck[ i ];
deck[ i ] = temp;
}
}
```

There are several other random number generators available in the standard library:

```
double drand48(void);
double erand48(unsigned short xsubi[3]);
long lrand48(void);
long nrand48(unsigned short xsubi[3]);
long mrand48(void);
long jrand48(unsigned short xsubi[3]);
void srand48(long seed);
unsigned short *seed48(unsigned short seed[3]);
void lcong48(unsigned short param[7]);
```

This family of functions generates uniformly distributed pseudo-random numbers.

Functions drand48() and erand48() return non-negative double-precision floating-point values uniformly distributed over the interval `[0.0, 1.0)`.

Functions lrand48() and nrand48() return non-negative long integers uniformly distributed over the interval `[0, 2**31)`.

Functions mrand48() and jrand48() return signed long integers uniformly distributed over the interval `[-2**31, 2**31)`.

Functions srand48(), seed48(), and lcong48() set the seeds for drand48(), lrand48(), or mrand48() and one of these should be called first.

Further examples of using these functions is given is Chapter 20.

15.3 String Conversion

There are a few functions that exist to convert strings to integer, long integer and float values. They are:

`double atof(char *string)` — Convert string to floating point value.
`int atoi(char *string)` — Convert string to an integer value
`int atol(char *string)` — Convert string to a long integer value.
`double strtod(char *string, char  *endptr)` — Convert string to a floating point value.
`long strtol(char *string, char  *endptr, int radix)` — Convert string to a long integer using a given radix.
`unsigned long strtoul(char  *string, char *endptr, int radix)` — Convert string to unsigned long.

Most of these are fairly straightforward to use. For example:

```
char *str1 = "100";
char *str2 = "55.444";
char *str3 = "      1234";
char *str4 = "123four";
char *str5 = "invalid123";

int i;
float f;

i = atoi(str1);  /* i = 100 */
f = atof(str2);  /* f = 55.44 */
i = atoi(str3);  /* i = 1234 */
i = atoi(str4);  /* i = 123 */
i = atoi(str5);  /* i = 0 */
```

Note:

- Leading blank characters are skipped.
- Trailing illegal characters are ignored.
- If conversion cannot be made zero is returned and `errno` (See Chapter 17) is set with the value `ERANGE`.

15.4 Searching and Sorting

The `stdlib.h` provides 2 useful functions to perform general searching and sorting of data on any type. In fact we have already introduced the `qsort()` function in Chapter 11.3. For completeness we list the prototype again here but refer the reader to the previous Chapter for an example.

The `qsort` standard library function is very useful function that is designed to sort an array by a *key* value of *any type* into ascending order, as long as the elements of the array are of fixed type.

qsort is prototyped (in `stdlib.h`):

```
void qsort(void *base, size_t num_elements, size_t element_size,
   int (*compare)(void const *, void  const *));
```

Similarly, there is a binary search function, `bsearch()` which is prototyped (in `stdlib.h`) as:

```
void *bsearch(const void *key, const void *base, size_t nel,
     size_t size, int (*compare)(const  void  *,  const  void *));
```

Using the same `Record` structure and `record_compare` function as the `qsort()` example (in Chapter 11.3):

```
typedef struct {
        int   key;
struct other_data;
} Record;

int record\_compare(void const *a, void  const *a)
  {  return ( ((Record *)a)->key - ((Record *)b)->key );
  }
```

Also, Assuming that we have an `array` of `array_length Record`s suitably filled with date we can call `bsearch()` like this:

```
Record key;
Record *ans;

key.key =  3; /* index value to be searched for */
ans = bsearch(&key, array, arraylength, sizeof(Record), record_compare);
```

The function `bsearch()` return a pointer to the field whose key filed is filled with the matched value of `NULL` if no match found.

Note that the type of the `key` argument **must** be the same as the array elements (`Record` above), even though only the `key.key` element is required to be set.

15.5 Exercises

Exercise 15.1 *Write a program that simulates throwing a six sided die*

Exercise 15.2 *Write a program that simulates the UK National lottery by selecting six different whole numbers in the range 1 – 49.*

Exercise 15.3 *Write a program that read a number from command line input and generates a random floating point number in the range 0 – the input number.*

Chapter 16

Mathematics: <math.h>

Mathematics is relatively straightforward library to use again. You **must** `#include <math.h>` and must **remember** to link in the math library at compilation:

```
    cc mathprog.c -o mathprog -lm
```

A common source of error is in forgetting to include the <`math.h`> file (and yes experienced programmers make this error also). Unfortunately the C compiler does not help much. Consider:

```
double x;
x = sqrt(63.9);
```

Having not seen the prototype for `sqrt` the compiler (by default) assumes that the function returns an `int` and converts the value to a `double` with meaningless results.

16.1 Math Functions

Below we list some common math functions. Apart from the note above they should be easy to use and we have already used some in previous examples. We give no further examples here:

`double acos(double x)` — Compute arc cosine of x.
`double asin(double x)` — Compute arc sine of x.
`double atan(double x)` — Compute arc tangent of x.
`double atan2(double y, double x)` — Compute arc tangent of y/x.
`double ceil(double x)` — Get smallest integral value that exceeds x.

`double cos(double x)` — Compute cosine of angle in radians.
`double cosh(double x)` — Compute the hyperbolic cosine of x.
`div_t div(int number, int denom)` — Divide one integer by another.
`double exp(double x` — Compute exponential of x
`double fabs (double x )` — Compute absolute value of x.
`double floor(double x)` — Get largest integral value less than x.
`double fmod(double x, double y)` — Divide x by y with integral quotient and return remainder.
`double frexp(double x, int *expptr)` — Breaks down x into mantissa and exponent of no.
`labs(long n)` — Find absolute value of long integer n.
`double ldexp(double x, int exp)` — Reconstructs x out of mantissa and exponent of two.
`ldiv_t ldiv(long number, long denom)` — Divide one long integer by another.
`double log(double x)` — Compute log(x).
`double log10 (double x )` — Compute log to the base 10 of x.
`double modf(double x, double *intptr)` — Breaks x into fractional and integer parts.
`double pow (double x, double y)` — Compute x raised to the power y.
`double sin(double x)` — Compute sine of angle in radians.
`double sinh(double x)` – Compute the hyperbolic sine of x.
`double sqrt(double x)` — Compute the square root of x.
`void srand(unsigned seed)` — Set a new seed for the random number generator (rand).
`double tan(double x)` — Compute tangent of angle in radians.
`double tanh(double x)` — Compute the hyperbolic tangent of x.

16.2 Math Constants

The `math.h` library defines many (often neglected) constants. It is always advisable to use these definitions:

`HUGE` — The maximum value of a single-precision floating-point number.

`M_E` — The base of natural logarithms (e).

`M_LOG2E` — The base-2 logarithm of e.

`M_LOG10E` – The base-10 logarithm of e.

`M_LN2` — The natural logarithm of 2.

`M_LN10` — The natural logarithm of 10.

`M_PI` — π.

`M_PI_2` — $\pi/2$.

`M_PI_4` — $\pi/4$.

`M_1_PI` — $1/\pi$.

`M_2_PI` — $2/\pi$.

`M_2_SQRTPI` — $2/\sqrt{\pi}$.

`M_SQRT2` — The positive square root of 2.

`M_SQRT1_2` — The positive square root of 1/2.

`MAXFLOAT` — The maximum value of a non-infinite single- precision floating point number.

`HUGE_VAL` — positive infinity.

There are also a number a machine dependent values defined in `#include <value.h>` — see `man value` or list `value.h` for further details.

Chapter 17

Input and Output (I/O):`stdio.h`

This chapter will look at many forms of I/O. We have briefly mentioned some forms before will look at these in much more detail here.

Your programs will need to include the standard I/O *header* file so do:

```
#include <stdio.h>
```

17.1 Reporting Errors

Many times it is useful to report errors in a C program. The standard library `perror()` is an easy to use and convenient function. It is used in conjunction with `errno` and frequently on encountering an error you may wish to terminate your program early. Whilst not strictly part of the `stdio.h` library we introduce the concept of `errno` and the function `exit()` here. We will meet these concepts in other parts of the Standard Library also.

17.1.1 `perror()`

The function `perror()` is prototyped by:

```
void perror(const char *message);
```

perror() produces a message (on standard error output — see Section 17.2.1), describing the last error encountered, returned to `errno` (see below) during a call to a system or library function. The argument string `message` is printed first, then a colon and a blank, then the message and a newline. If `message` is a NULL pointer or points to a null string, the colon is not printed.

17.1.2 errno

`errno` is a special system variable that is set if a system call cannot perform its set task. It is defined in `#include <errno.h>`.

To use `errno` in a C program it must be declared via:

```
extern int errno;
```

It can be manually reset within a C program (although this is uncommon practice) otherwise it simply retains its last value returned by a system call or library function.

17.1.3 `exit()`

The function `exit()` is prototyped in `#include <stdlib>` by:

```
void exit(int status)
```

Exit simply terminates the execution of a program and returns the exit `status` value to the operating system. The `status` value is used to indicate if the program has terminated properly:

- it exist with a `EXIT_SUCCESS` value on successful termination
- it exist with a `EXIT_FAILURE` value on unsuccessful termination.

On encountering an error you may frequently call an `exit(EXIT_FAILURE)` to terminate an errant program.

17.2 Streams

Streams are a portable way of reading and writing data. They provide a flexible and efficient means of I/O.

A Stream is a file or a physical device (*e.g.* printer or monitor) which is manipulated with a **pointer** to the stream.

There exists an internal C data structure, `FILE`, which represents all streams and is defined in `stdio.h`. We simply need to refer to the `FILE` structure in C programs when performing I/O with streams.

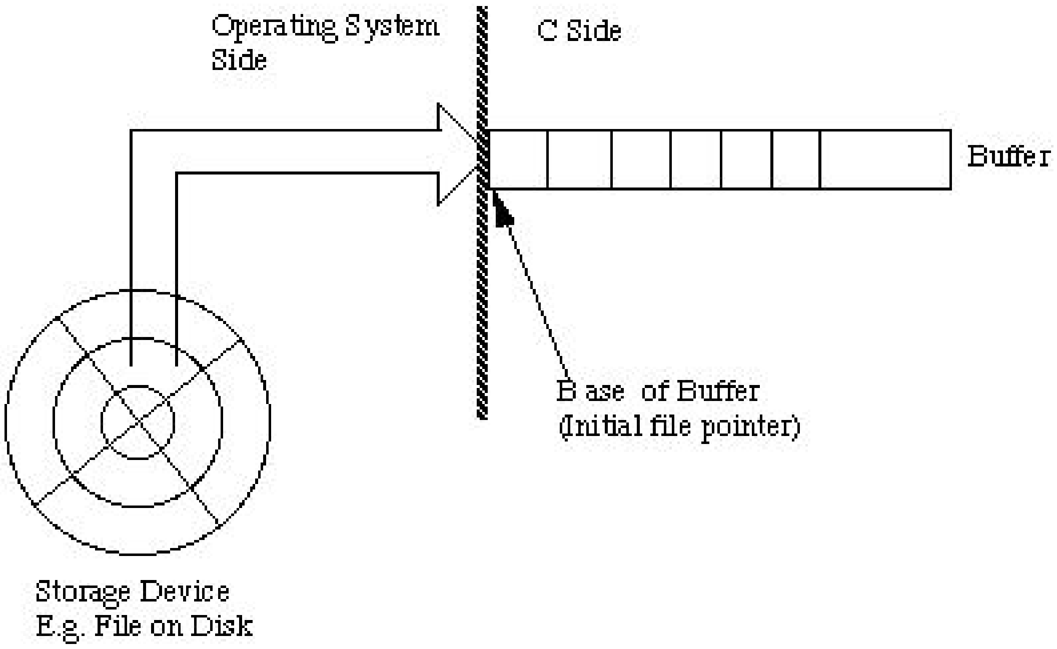

Operating System Side
C Side
Buffer
Base of Buffer (Initial file pointer)
Storage Device E.g. File on Disk

Figure 17.1: Stream I/O Model

We just need to declare a variable or pointer of this type in our programs.

We do not need to know any more specifics about this definition.

We must open a stream before doing any I/O,

then access it

and then close it.

Stream I/O is **BUFFERED**: That is to say a fixed "chunk" is read from or written to a file via some temporary storage area (the buffer). This is illustrated in Fig. 17.1. NOTE the file pointer actually points to this buffer.

This leads to efficient I/O but **beware**: data written to a buffer does not appear in a file (or device) until the buffer is flushed or written out. (\n does this). Any abnormal exit of code can cause problems.

17.2.1 Predefined Streams

UNIX defines 3 predefined streams (in `stdio.h`):

`stdin, stdout, stderr`

They all use text a the method of I/O.

`stdin` and `stdout` can be used with files, programs, I/O devices such as keyboard, console, *etc.*. `stderr` always goes to the console or screen.

The console is the default for `stdout` and `stderr`. The keyboard is the default for `stdin`.

Predefined stream are automatically open.

Redirection

This how we override the UNIX default predefined I/O defaults.

This is not part of C but operating system dependent. We will do redirection from the command line.

> — redirect `stdout` to a file.

So if we have a program, `out`, that usually prints to the screen then

```
out > file1
```

will send the output to a file, `file1`.

< — redirect `stdin` from a file to a program.

So if we are expecting input from the keyboard for a program, `in` we can read similar input from a file

```
in < file2.
```

| — *pipe*: puts `stdout` from one program to `stdin` of another

```
prog1 | prog2
```

e.g. Sent output (usually to console) of a program direct to printer:

```
out | lpr
```

17.3 Basic I/O

There are a couple of function that provide basic I/O facilities.

probably the most common are: `getchar()` and `putchar()`. They are defined and used as follows:

- `int getchar(void)` — reads a char from `stdin`
- `int putchar(char ch)` — writes a char to `stdout`, returns character written.

```
int ch;

ch = getchar();
(void) putchar((char) ch);
```

Related Functions:

```
  int getc(FILE *stream),
int putc(char ch,FILE *stream)
```

17.4 Formatted I/O

We have seen examples of how C uses formatted I/O already. Let's look at this in more detail.

17.4.1 Printf

The function is defined as follows:

`int printf(char *format, arg list ...)` —
prints to `stdout` the list of arguments according specified format string. Returns number of characters printed.

The **format string** has 2 types of object:

- *ordinary characters* — these are copied to output.
- *conversion specifications* — denoted by % and listed in Table 17.1.

Between % and format char we can put:

Format Spec (%)	Type	Result
c	char	single character
i,d	int	decimal number
o	int	octal number
x,X	int	hexadecimal number lower/uppercase notation
u	int	unsigned int
s	char *	print string terminated by \0
f	double/float	format -m.ddd...
e,E	”	Scientific Format -1.23e002
g,G	”	e or f whichever is most compact
%	–	print % character

Table 17.1: Printf/scanf format characters

- (minus sign) — left justify.

integer number — field width.

m.d — m = field width, d = precision of number of digits after decimal point or number of chars from a string.

So:

```
printf("%-2.3f\n",17.23478);
```

The output on the screen is:

```
17.235
```

and:

```
        printf("VAT=17.5%%\n");
```

...outputs:

```
        VAT=17.5%
```

17.5 scanf

This function is defined as follows:

`int scanf(char *format, args....)` — reads from stdin and puts input in address of variables specified in `args` list. Returns number of chars read.

Format control string similar to `printf`

Note: The ADDRESS of variable or a pointer to one is required by `scanf`.

```
  scanf(``%d'',&i);
```

We can just give the name of an array or string to scanf since this corresponds to the start address of the array/string.

```
char string[80];
scanf(``%s'',string);
```

17.6 Files

Files are the most common form of a stream.

The first thing we must do is *open* a file. The function `fopen()` does this:

```
  FILE *fopen(char *name, char *mode)
```

`fopen` returns a pointer to a FILE. The `name` string is the name of the file on disc that we wish to access. The `mode` string controls our type of access. If a file cannot be accessed for any reason a `NULL` pointer is returned.

Modes include: "r" — read,
"w" — write and
"a" — append.

To open a file we must have a stream (file pointer) that *points* to a `FILE` structure.

So to open a file, called *myfile.dat* for reading we would do:

```
  FILE *stream, *fopen();
  /* declare a stream and prototype fopen */

  stream = fopen(``myfile.dat'',``r'');
```

it is good practice to to check file is opened

```
  if ( (stream = fopen( ``myfile.dat'',
                        ``r'')) == NULL)
     {  printf(``Can't open %s\n'',
                        ``myfile.dat'');
       exit(1);
     }
  ......
```

17.6.1 Reading and writing files

The functions `fprintf` and `fscanf` a commonly used to access files.

```
int fprintf(FILE *stream, char *format, args..)
int fscanf(FILE *stream, char *format, args..)
```

These are similar to `printf` and `scanf` except that data is read from the *stream* that must have been opened with `fopen()`.

The `stream` pointer is automatically incremented with ALL file read/write functions. We **do not** have to worry about doing this.

```
char *string[80]
FILE *stream, *fopen();

if ( (stream = fopen(...))  != NULL)
   fscanf(stream,``%s'', string);
```

Other functions for files:

```
int getc(FILE *stream), int fgetc(FILE *stream)
int putc(char ch, FILE *s), int fputc(char ch, FILE *s)
```

These are like `getchar`, `putchar`.

getc is defined as preprocessor MACRO in `stdio.h`. fgetc is a C library function. Both achieve the same result!!

`fflush(FILE *stream)` — flushes a stream.
`fclose(FILE *stream)` — closes a stream.

We can access predefined streams with `fprintf` *etc.*

```
fprintf(stderr,``Cannot Compute!!\n'');
  fscanf(stdin,``%s'',string);
```

17.7 sprintf and sscanf

These are like `fprintf` and `fscanf` except they read/write to a string.

```
int sprintf(char *string, char *format, args..)
int sscanf(char *string, char *format, args..)
```

For Example:

float full_tank = 47.0; /* litres */
float miles = 300;
char miles_per_litre[80];

sprintf(miles_per_litre,“Miles per litre
= %2.3f”, miles/full_tank);

17.7.1 Stream Status Enquiries

There are a few useful stream enquiry functions, prototyped as follows:

```
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
int fileno(FILE *stream);
```

Their use is relatively simple:

`feof()` — returns true if the stream is currently at the end of the file. So to read a stream,`fp`, line by line you could do:

```
while ( !feof(fp) )
  fscanf(fp,"%s",line);
```

`ferror()` — reports on the error state of the stream and returns true if an error has occurred.

`clearerr()` — resets the error indication for a given stream.

`fileno()` — returns the integer file descriptor associated with the named stream.

17.8 Low Level I/O

This form of I/O is UNBUFFERED — each read/write request results in accessing disk (or device) directly to fetch/put a specific number of **bytes**.

There are no formatting facilities — we are dealing with bytes of information.

This means we are now using binary (and not text) files.

Instead of file pointers we use *low level* `file handle` or `file descriptors` which give a unique integer number to identify each file.

To Open a file use:

`int open(char *filename, int flag, int perms)` — this returns a file descriptor or -1 for a **fail**.

The `flag` controls file access and has the following predefined in `fcntl.h`:

`O_APPEND, O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_WRONLY` + others see online `man` pages or reference manuals.

`perms` — best set to 0 for most of our applications.

The function:

`creat(char *filename, int perms)`

can also be used to create a file.

`int close(int handle)` — close a file

`int read(int handle, char *buffer,`
`unsigned length)`

`int write(int handle, char *buffer, unsigned length)`

are used to read/write a specific number of bytes from/to a file (handle) stored or to be put in the memory location specified by `buffer`.

The `sizeof()` function is commonly used to specify the `length`.

read and write return the number of bytes read/written or -1 if they fail.

```
/* program to read a list of floats from a binary file */
/* first byte of file is an integer saying how many */
/* floats in file. Floats follow after it, File name got from */
/* command line */

#include<stdio.h>
#include<fcntl.h>

float bigbuff[1000];
```

```
main(int argc, char **argv)

{ int fd;
  int bytes_read;
  int file_length;

  if ( (fd = open(argv[1],O_RDONLY)) = -1)
      { /* error file not open */....
        perror("Datafile");
        exit(1);
      }
  if ( (bytes_read = read(fd,&file_length,
        sizeof(int))) == -1)
      { /* error reading file */...
        exit(1);
      }
  if ( file_length > 999 ) {/* file too big */ ....}
  if ( (bytes_read = read(fd,bigbuff,
        file_length*sizeof(float))) == -1)
      { /* error reading open */...
        exit(1);
      }
}
```

17.9 Exercises

Exercise 17.1 *Write a program to copy one named file into another named file. The two file names are given as the first two arguments to the program.*

Copy the file a block (512 bytes) at a time.

```
Check:  that the program has two arguments
         or print "Program need two arguments"
     that the first name file is readable
         or print "Cannot open file .... for reading"
     that the second file is writable
         or print "Cannot open file .... for writing"
```

Exercise 17.2 *Write a program last that prints the last n lines of a text file, by n and the file name should be specified form command line input. By default n should be 5, but your program should allow an optional argument so that*

```
last -n file.txt
```

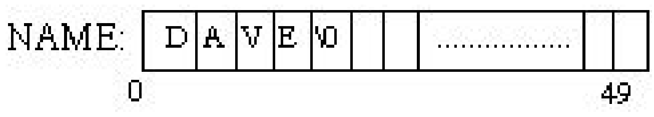
prints out the last n lines, where n is any integer. Your program should make the best use of available storage.

Exercise 17.3 *Write a program to compare two files and print out the lines where they differ. Hint: look up appropriate string and file handling library routines. This should not be a very long program.*

Chapter 18

String Handling: <`string.h`>

Recall from our discussion of arrays (Chapter 6) that strings are defined as an array of characters or a pointer to a portion of memory containing ASCII characters. A `string` in C is a sequence of zero or more characters followed by a `NULL` (\0) character:



It is important to preserve the `NULL` terminating character as it is how C defines and manages variable length strings. **All** the C standard library functions require this for successful operation.

In general, apart from some *length-restricted functions* (`strncat(), strncmp,()` and `strncpy()`), unless you create strings by hand you should not encounter any such problems, . You should use the many useful string handling functions and not really need to *get your hands dirty* dismantling and assembling strings.

18.1 Basic String Handling Functions

All the string handling functions are prototyped in:

`#include <string.h>`

The common functions are described below:

`char *stpcpy (const char *dest,const char  *src)` — Copy one string into another.

`int strcmp(const char *string1,const char *string2)` – Compare string1 and string2 to determine alphabetic order.
`char *strcpy(const char *string1,const char *string2)` — Copy string2 to stringl.
`char *strerror(int errnum)` — Get error message corresponding to specified error number.
`int strlen(const char *string)` — Determine the length of a string.
`char *strncat(const char *string1, char *string2, size_t n)` — Append n characters from string2 to stringl.
`int strncmp(const char *string1, char *string2, size_t n)` — Compare first n characters of two strings.
`char *strncpy(const char *string1,const char *string2, size_t n)` — Copy first n characters of string2 to stringl .
`int strcasecmp(const char *s1, const char *s2)` — case insensitive version of `strcmp()`.
`int strncasecmp(const char *s1, const char *s2, int n)` — case insensitive version of `strncmp()`.

The use of most of the functions is straightforward, for example:

```
char *str1 = "HELLO";
char *str2;
int length;

length = strlen("HELLO"); /* length = 5 */
(void) strcpy(str2,str1);
```

Note that both `strcat()` and `strcopy()` both return a copy of their first argument which is the destination array. Note the order of the arguments is *destination array* followed by *source array* which is sometimes easy to get the wrong around when programming.

The `strcmp()` function *lexically* compares the two input strings and returns:

Less than zero — if `string1` is lexically less than `string2`

Zero — if `string1` and `string2` are lexically equal

Greater than zero — if `string1` is lexically greater than `string2`

This can also confuse beginners and experience programmers forget this too.

The `strncat()`, `strncmp,()` and `strncpy()` copy functions are string restricted version of their more general counterparts. They perform a similar task but only up to the first `n` characters. Note the the `NULL` terminated requirement may get violated when using these functions, for example:

```
char *str1 = "HELLO";
char *str2;
int length = 2;


(void) strcpy(str2,str1, length); /* str2 = "HE" */
```

`str2` **is NOT NULL TERMINATED!! — BEWARE**

18.1.1 String Searching

The library also provides several string searching functions:

`char *strchr(const char *string, int c)` — Find first occurrence of character `c` in string.
`char *strrchr(const char *string, int c)` — Find last occurrence of character `c` in string.
`char *strstr(const char *s1, const char *s2)` — locates the first occurrence of the string `s2` in string `s1`.
`char *strpbrk(const char *s1, const char *s2)` — returns a pointer to the first occurrence in string s1 of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`
`size_t strspn(const char *s1, const char *s2)` — returns the number of characters at the begining of `s1` that match `s2`.
`size_t strcspn(const char *s1, const char *s2)` — returns the number of characters at the begining of `s1` that *do not* match `s2`.
`char *strtok(char *s1, const char *s2)` — break the string pointed to by `s1` into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by `s2`.
`char *strtok_r(char *s1, const char *s2, char **lasts)` — has the same functionality as strtok() except that a pointer to a string placeholder lasts must be supplied by the caller.

`strchr()` and `strrchr()` are the simplest to use, for example:

```
char *str1 = "Hello";
char *ans;

ans = strchr(str1,'l');
```

After this execution, `ans` points to the location `str1 + 2`

`strpbrk()` is a more general function that searches for the first occurrence of any of a group of characters, for example:

```
char *str1 = "Hello";
char *ans;

ans = strpbrk(str1,'aeiou');
```

Here, `ans` points to the location `str1 + 1`, the location of the first `e`.

strstr() returns a pointer to the specified search string or a null pointer if the string is not found. If s2 points to a string with zero length (that is, the string ""), the function returns s1. For example,

```
char *str1 = "Hello";
char *ans;

ans = strstr(str1,'lo');
```

will yield `ans` = `str + 3`.

`strtok()` is a little more complicated in operation. If the first argument is not NULL then the function finds the position of any of the second argument characters. However, the position is remembered and any subsequent calls to `strtok()` will start from this position if on these subsequent calls the first argument is `NULL`. For example, If we wish to break up the string `str1` at each space and print each token on a new line we could do:

```
char *str1 = "Hello Big Boy";
char *t1;


for ( t1 = strtok(str1," ");
      t1 != NULL;
      t1 = strtok(NULL, " ") )

printf("%s\n",t1);
```

Here we use the for loop in a non-standard counting fashion:

- The initialisation calls `strtok()` loads the function with the string `str1`
- We terminate when t1 is `NULL`
- We keep assigning tokens of `str1` to `t1` until termination by calling `strtok()` with a `NULL` first argument.

18.2 Character conversions and testing: `ctype.h`

We conclude this chapter with a related library `#include <ctype.h>` which contains many useful functions to convert and test *single* characters. The common functions are prototypes as follows:

Character testing:

`int isalnum(int c)` — True if c is alphanumeric.
`int isalpha(int c)` — True if c is a letter.
`int isascii(int c)` — True if c is ASCII .
`int iscntrl(int c)` — True if c is a control character.
`int isdigit(int c)` — True if c is a decimal digit
`int isgraph(int c)` — True if c is a graphical character.
`int islower(int c)` — True if c is a lowercase letter
`int isprint(int c)` — True if c is a printable character
`int ispunct (int c)` — True if c is a punctuation character.
`int isspace(int c)` — True if c is a space character.
`int isupper(int c)` — True if c is an uppercase letter.
`int isxdigit(int c)` — True if c is a hexadecimal digit

Character Conversion:

`int toascii(int c)` — Convert `c` to ASCII .
`tolower(int c)` — Convert c to lowercase.
`int toupper(int c)` — Convert c to uppercase.

The use of these functions is straightforward and we do not give examples here.

18.3 Memory Operations: <`memory.h`>

Although not strictly string functions the functions are prototyped in `#include <string.h>`:

`void *memchr (void *s, int c, size_t n)` — Search for a character in a buffer .
`int memcmp (void *s1,  void  *s2, size_t n)` — Compare two buffers.
`void *memcpy (void *dest,  void  *src, size_t n)` — Copy one buffer into another .
`void *memmove (void *dest,  void  *src, size_t n)` — Move a number of bytes from one buffer lo another.
`void *memset (void *s, int c, size_t n)` — Set all bytes of a buffer to a given character.

Their use is fairly straightforward and not dissimilar to comparable string operations (except the exact length (`n`) of the operations must be specified as there is no natural termination here).

Note that in all case to **bytes** of memory are copied. The `sizeof()` function comes in handy again here, for example:

```
char src[SIZE],dest[SIZE];
int  isrc[SIZE],idest[SIZE];

/* Copy chars (bytes) ok */
memcpy(dest,src, SIZE);

/* Copy arrays of ints */
memcpy(idest,isrc, SIZE*sizeof(int));
```

`memmove()` behaves in exactly the same way as `memcpy()` except that the source and destination locations may overlap.

`memcmp()` is similar to `strcmp()` except here *unsigned bytes* are compared and returns less than zero if `s1` is less than `s2` *etc.*

18.4 Exercises

Exercise 18.1 *Write a function similar to strlen that can handle unterminated strings. Hint: you will need to know and pass in the length of the string.*

Exercise 18.2 *Write a function that returns true if an input string is a palindrome of each other. A palindrome is a word that reads the same backwards as it does forwards* e.g *ABBA.*

Exercise 18.3 *Suggest a possible implementation of the* `strtok()` *function:*

1. *using other string handling functions.*
2. *from first pointer principles*

How is the storage of the tokenised string achieved?

Exercise 18.4 *Write a function that converts all characters of an input string to upper case characters.*

Exercise 18.5 *Write a program that will reverse the contents stored in memory in bytes. That is to say if we have n bytes in memory byte n becomes byte 0, byte $n-1$ becomes byte 1* etc.

Chapter 19

File Access and Directory System Calls

There are many UNIX utilities that allow us to manipulate directories and files. `cd, ls, rm, cp, mkdir` *etc.* are examples we have (hopefully) already met.

We will now see how to achieve similar tasks from within a C program.

19.1 Directory handling functions: <`unistd.h`>

This basically involves calling appropriate functions to traverse a directory hierarchy or inquire about a directories contents.

`int chdir(char *path)` — changes directory to specified path string.

Example: C emulation of UNIX's `cd` command:

```
#include<stdio.h>
#include<unistd.h>

main(int argc,char **argv)
   {
     if (argc < 2)
        {  printf(``Usage:  %s
              <pathname> \n'',argv[0]);
           exit(1);
        }
     if (chdir(argv[1]) != 0)
        { printf(``Error in chdir\n'');
          exit(1);
        }
   }
```

`char *getwd(char *path)` — get the full pathname of the current working directory. `path` is a pointer to a string where the pathname will be returned. `getwd` returns a pointer to the string or NULL if an error occurs.

19.1.1 Scanning and Sorting Directories:`<sys/types.h>`,`<sys/dir`

Two useful functions (On BSD platforms and **NOT** in multi-threaded application) are available

`scandir(char *dirname, struct direct **namelist, int (*select)(), int (*compar)())` — reads the directory dirname and builds an array of pointers to directory entries or -1 for an error. `namelist` is a pointer to an array of structure pointers.

`(*select))()` is a pointer to a function which is called with a pointer to a directory entry (defined in <sys/types> and should return a non zero value if the directory entry should be included in the array. If this pointer is NULL, then all the directory entries will be included.

The last argument is a pointer to a routine which is passed to `qsort` (see `man qsort`) — a built in function which sorts the completed array. If this pointer is NULL, the array is not sorted.

`alphasort(struct direct **d1, **d2)` — alphasort() is a built in routine which will sort the array alphabetically.

Example - a simple C version of UNIX `ls` utility

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
```

```
#include <stdio.h>

#define FALSE 0
#define TRUE !FALSE

extern int alphasort();

char pathname[MAXPATHLEN];

main()    { int count,i;
            struct direct **files;
            int file_select();

            if (getwd(pathname) == NULL )
               { printf("Error getting path\n");
                 exit(0);
               }
            printf("Current Working Directory = %s\n",pathname);
            count =
              scandir(pathname, &files, file_select, alphasort);

            /* If no files found, make a non-selectable menu item */
            if (count <= 0)
               { printf(``No files in this directory\n'');
                 exit(0);
               }
            printf(``Number of files = %d\n'',count);
            for (i=1;i<count+1;++i)
               printf(``%s '',files[i-1]- >d_name);
            printf(``\n''); /* flush buffer */
          }



int file_select(struct direct *entry)

    {if ((strcmp(entry- >d_name, ``.'')  == 0) ||
          (strcmp(entry- >d_name, ``..'')  == 0))
```

```
            return (FALSE);
     else
            return (TRUE);
    }
```

scandir returns the current directory (.) and the directory above this (..) as well as all files so we need to check for these and return FALSE so that they are not included in our list.

Note: scandir and alphasort have definitions in sys/types.h and sys/dir.h.
MAXPATHLEN and getwd definitions in sys/param.h

We can go further than this and search for specific files: Let's write a modified
file_select() that only scans for files with a .c, .o or .h suffix:

```
int file_select(struct direct *entry)

    {char *ptr;
     char *rindex(char *s, char c);

     if ((strcmp(entry->d_name, ``.'')  == 0) ||
          (strcmp(entry->d_name, ``..'')  == 0))
            return (FALSE);

     /* Check for filename extensions */
     ptr = rindex(entry->d_name, '.')
     if ((ptr != NULL) &&
          ((strcmp(ptr, ``.c'') == 0)
          || (strcmp(ptr, ``.h'') == 0)
          || (strcmp(ptr, ``.o'') == 0) ))
            return (TRUE);
     else
          return(FALSE);
    }
```

NOTE: rindex() is a string handling function that returns a pointer to the last occurrence of character c in string s, or a NULL pointer if c does not occur in the string. (`index()` is similar function but assigns a pointer to 1st occurrence.)

The function `struct direct *readdir(char *dir)` also exists in <sys/dir.h>¿ to return a given directory `dir` listing.

19.2 File Manipulation Routines: unistd.h, sys/types.h, sys/stat.h

There are many system calls that can applied directly to files stored in a directory.

19.2.1 File Access

`int access(char *path, int mode)` — determine accessibility of file.

`path` points to a path name naming a file. `access()` checks the named file for accessibility according to `mode`, defined in `#include <unistd.h>`:

R_OK – test for read permission

W_OK – test for write permission

X_OK – test for execute or search permission

F_OK – test whether the directories leading to the file can be searched and the file exists.

`access()` returns: 0 on success, -1 on failure and sets `errno` to indicate the error. See `man` pages for list of errors.

errno

`errno` is a special system variable that is set if a system call cannot perform its set task.

To use `errno` in a C program it must be declared via:

```
extern int errno;
```

It can be manually reset within a C program other wise it simply retains its last value.

`int chmod(char *path, int mode)` change the mode of access of a file. specified by `path` to the given `mode`.

`chmod()` returns 0 on success, -1 on failure and sets `errno` to indicate the error. Errors are defined in `#include <sys/stat.h>`

The access mode of a file can be set using predefined macros in `sys/stat.h` — see `man` pages — or by setting the mode in a a 3 digit octal number.

The rightmost digit specifies owner privileges, middle group privileges and the leftmost other users privileges.

For each octal digit think of it a 3 bit binary number. Leftmost bit = read access (on/off) middle is write, right is executable.

So 4 (octal 100) = read only, 2 (010) = write, 6 (110) = read and write, 1 (001) = execute.

so for access mode 600 gives user read and write access others no access. 666 gives everybody read/write access.

NOTE: a UNIX command `chmod` also exists

19.2.2 File Status

Two useful functions exist to inquire about the files current status. You can find out how large the file is (`st_size`) when it was created (`st_ctime`) *etc.* (see `stat` structure definition below. The two functions are prototyped in `<sys/stat.h>`

```
int stat(char *path, struct stat *buf),
int fstat(int fd, struct
stat *buf)
```

`stat()` obtains information about the file named by path. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

`fstat()` obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an `open` call (Low level I/O).

`stat(),` and `fstat()` return 0 on success, -1 on failure and sets `errno` to indicate the error. Errors are again defined in `#include <sys/stat.h>`

`buf` is a pointer to a stat structure into which information is placed concerning the file. A stat structure is define in `#include <sys/types.h>`, as follows

```
struct stat {
          mode_t   st_mode;     /* File mode (type, perms) */
          ino_t    st_ino;      /* Inode number */
          dev_t    st_dev;      /* ID of device containing */
                                /* a directory entry for this file */
          dev_t    st_rdev;     /* ID of device */
                                /* This entry is defined only for */
                                /* char special or block special files */
          nlink_t  st_nlink;    /* Number of links */
          uid_t    st_uid;      /* User ID of the file's owner */
          gid_t    st_gid;      /* Group ID of the file's group */
          off_t    st_size;     /* File size in bytes */
          time_t   st_atime;    /* Time of last access */
          time_t   st_mtime;    /* Time of last data modification */
          time_t   st_ctime;    /* Time of last file status change */
                                /* Times measured in seconds since */
                                /* 00:00:00 UTC, Jan. 1, 1970 */
          long     st_blksize;  /* Preferred I/O block size */
          blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
}
```

19.2.3 File Manipulation:stdio.h, unistd.h

There are few functions that exist to delete and rename files. Probably the most common way is to use the `stdio.h` functions:

```
int remove(const char *path);
int rename(const char *old, const char *new);
```

Two system calls (defined in `unistd.h`) which are actually used by `remove()` and `rename()` also exist but are probably harder to remember unless you are familiar with UNIX.

`int unlink(cons char *path)` — removes the directory entry named by `path` `unlink()` returns 0 on success, -1 on failure and sets `errno` to indicate the error. Errors listed in `#include <sys/stat.h>`

A similar function `link(const char *path1, const char *path2)` creates a linking from an existing directory entry `path1` to a new entry `path2`

19.2.4 Creating Temporary FIles:<stdio.h>

Programs often need to create files just for the life of the program. Two convenient functions (plus some variants) exist to assist in this task. Management (deletion of files etc) is taken care of by the Operating System.

The function `FILE *tmpfile(void)` creates a temporary file and opens a corresponding stream. The file will automatically be deleted when all references to the file are closed.

The function `char *tmpnam(char *s)` generate file names that can safely be used for a temporary file. Variant functions `char *tmpnam_r(char *s)` and `char *tempnam(const char *dir, const char *pfx)` also exist

NOTE: There are a few more file manipulation routines not listed here see `man` pages.

19.3 Exercises

Exercise 19.1 *Write a C program to emulate the* `ls -l` *UNIX command that prints all files in a current directory and lists access privileges etc. DO NOT simply* `exec ls -l` *from the program.*

Exercise 19.2 *Write a program to print the lines of a file which contain a word given as the program argument (a simple version of* `grep` *UNIX utility).*

Exercise 19.3 *Write a program to list the files given as arguments, stopping every 20 lines until a key is hit.(a simple version of* `more` *UNIX utility)*

Exercise 19.4 *Write a program that will list all files in a current directory and all files in subsequent sub directories.*

Exercise 19.5 *Write a program that will only list subdirectories in alphabetical order.*

Exercise 19.6 *Write a program that shows the user all his/her C source programs and then prompts interactively as to whether others should be granted read permission; if affirmative such permission should be granted.*

Exercise 19.7 *Write a program that gives the user the opportunity to remove any or all of the files in a current working directory. The name of the file should appear followed by a prompt as to whether it should be removed.*

Chapter 20

Time Functions

In this chapter we will look at how we can access the clock time with UNIX system calls.

There are many more time functions than we consider here - see `man` pages and standard library function listings for full details. In this chapter we concentrate on applications of timing functions in C

Uses of time functions include:

- telling the time.
- timing programs and functions.
- setting number seeds.

20.1 Basic time functions

Some of thge basic time functions are prototypes as follows:

`time_t time(time_t *tloc)` — returns the time since `00:00:00 GMT, Jan. 1, 1970`, measured in seconds.

If `tloc` is not NULL, the return value is also stored in the location to which tloc points.

`time()` returns the value of time on success.

On failure, it returns `(time_t) -1`. `time_t` is typedefed to a long (int) in <sys/types.h> and <sys/time.h> header files.

`int ftime(struct timeb *tp)` — fills in a structure pointed to by `tp`, as defined in <sys/timeb.h>:

```
struct timeb
   { time_t time;
     unsigned short millitm;
     short timezone;
     short dstflag;
   };
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Day light Saving time applies locally during the appropriate part of the year.

On success, `ftime()` returns no useful value. On failure, it returns -1.

Two other functions defined *etc.* in #include <time.h>

`char *ctime(time_t *clock)`,
`char *asctime(struct tm *tm)`

`ctime()` converts a long integer, pointed to by `clock`, to a 26-character string of the form produced by `asctime()`. It first breaks down clock to a `tm` structure by calling `localtime()`, and then calls `asctime()` to convert that tm structure to a string.

`asctime()` converts a time value contained in a tm structure to a 26-character string of the form:

```
Sun Sep 16 01:03:52 1973
```

asctime() returns a pointer to the string.

20.2 Example time applications

we mentioned above three possible uses of time functions (there are many more) but these are very common.

20.2.1 Example 1: Time (in seconds) to perform some computation

This is a simple program that illustrates that calling the time function at distinct moments and noting the different times is a simple method of timing fragments of code:

```
/* timer.c */

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
  {  int i;
     time_t t1,t2;

     (void) time(&t1);
     for (i=1;i<=300;++i)
       printf(``%d %d %d\n'',i, i*i, i*i*i);
     (void) time(&t2);
     printf(``\n Time to do 300 squares and
     cubes= %d seconds\n'', (int) t2-t1);
  }
```

20.2.2 Example 2: Set a random number seed

We have seen a similar example previously, this time we use the `lrand48()` function to generate of number sequence:

```
/* random.c */
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
```

```
{ int i;
  time_t t1;

  (void) time(&t1);
  srand48((long) t1);
  /* use time in seconds to set seed */
  printf(``5 random numbers
     (Seed = %d):\n'',(int) t1);
  for (i=0;i<5;++i)
     printf(``%d '', lrand48());
  printf(``\n\n''); /* flush print buffer */
}
```

`lrand48()` returns non-negative long integers uniformly distributed over the interval (0, 2**31).

A similar function `drand48()` returns double precision numbers in the range [0.0,1.0).

`srand48()` sets the seed for these random number generators. It is important to have different seeds when we call the functions otherwise the same set of pseudo-random numbers will generated. `time()` always provides a unique seed.

20.3 Exercises

Exercise 20.1 *Write a C program that times a fragment of code in milliseconds.*

Exercise 20.2 *Write a C program to produce a series of floating point random numbers in the ranges (a) 0.0 - 1.0 (b) 0.0 - n where n is any floating point value. The seed should be set so that a unique sequence is guaranteed.*

Chapter 21

Process Control: `<stdlib.h>`,`<unistd.h>`

A *process* is basically a single running program. It may be a "system" program (*e.g* login, update, csh) or program initiated by the user (textedit, dbxtool or a user written one).

When UNIX runs a process it gives each process a unique number – a process ID, `pid`.

The UNIX command `ps` will list all current processes running on your machine and will list the pid.

The C function `int getpid()` will return the pid of process that called this function.

A program usually runs as a single process. However later we will see how we can make programs run as several separate communicating processes.

21.1 Running UNIX Commands from C

We can run commands from a C program just as if they were from the UNIX command line by using the `system()` function. **NOTE:** this can save us a lot of time and hassle as we can run other (proven) programs, scripts *etc.* to do set tasks.

`int system(char *string)` — where string can be the name of a unix utility, an executable shell script or a user program. System returns the exit status of the shell. System is prototyped in `<stdlib.h>`

Example: Call `ls` from a program

```
main()
{ printf(``Files in Directory are:\n'');
  system(``ls -l'');
}
```

`system` is a call that is made up of 3 other system calls: `execl(), wait()` and `fork()` (which are prototyed in <unistd>)

21.2 execl()

`execl` has 5 other related functions — see `man` pages.

`execl` stands for *execute* and *leave* which means that a process will get executed and then terminated by `execl`.

It is defined by:

```
execl(char *path, char *arg0,...,char *argn, 0);
```

The last parameter must always be 0. It is a *NULL terminator*. Since the argument list is variable we must have some way of telling C when it is to end. The NULL terminator does this job.

where `path` points to the name of a file holding a command that is to be executed, `argo` points to a string that is the same as path (or at least its last component.

`arg1 ... argn` are pointers to arguments for the command and 0 simply marks the end of the (variable) list of arguments.

So our above example could look like this also:

```
main()
{ printf(``Files in Directory are:\n'');
  execl(`/bin/ls'',``ls'', ``-l'',0);
}
```

21.3 fork()

`int fork()` turns a single process into 2 identical processes, known as the *parent* and the *child*. On success, fork() returns 0 to the child process and returns the process ID of the child process to the parent process. On failure, fork() returns -1 to the parent process, sets errno to indicate the error, and no child process is created.

NOTE: The child process will have its own unique PID.

The following program illustrates a simple use of fork, where two copies are made and run together (multitasking)

```
main()
{ int return_value;

  printf(``Forking process\n'');
  fork();
  printf(``The process id is %d
    and return value is %d\n",
    getpid(), return_value);
  execl(``/bin/ls/'',``ls'',``-l'',0);
  printf(``This line is not printed\n'');
}
```

The Output of this would be:

```
Forking process
The process id is 6753 and return value is 0
The process id is 6754 and return value is 0
two lists of files in current directory
```

NOTE: The processes have unique ID's which will be different at each run.

It also impossible to tell in advance which process will get to CPU's time — so one run may differ from the next.

When we spawn 2 processes we can easily detect (in each process) whether it is the child or parent since fork returns 0 to the child. We can trap any errors if fork returns a -1. *i.e.*:

```
int pid; /* process identifier */

pid = fork();
if ( pid < 0 )
   { printf(``Cannot fork!!\n'');
     exit(1);
   }
if ( pid == 0 )
   { /* Child process */ ......  }
else
   { /* Parent process pid is child's pid */
   ....  }
```

21.4 wait()

`int wait (int *status_location)` — will force a parent process to wait for a child process to stop or terminate. `wait()` return the pid of the child or -1 for an error. The exit status of the child is returned to `status_location`.

21.5 exit()

`void exit(int status)` — terminates the process which calls this function and returns the exit `status` value. Both UNIX and C (forked) programs can read the status value.

By convention, a status of 0 means *normal termination* any other value indicates an error or unusual occurrence. Many standard library calls have errors defined in the `sys/stat.h` header file. We can easily derive our own conventions.

A complete example of forking program is originally titled `fork.c`:

```
/* fork.c - example of a fork in a program */
/* The program asks for UNIX commands to be typed and inputted to a string*/
/* The string is then "parsed" by locating blanks etc. */
/* Each command and sorresponding arguments are put in a args array */
/* execvp is called to execute these commands in child process */
/* spawned by fork() */

/* cc -o fork fork.c */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
    char buf[1024];
    char *args[64];

    for (;;) {
        /*
         * Prompt for and read a command.
         */
        printf("Command: ");

        if (gets(buf) == NULL) {
            printf("\n");
            exit(0);
        }

        /*
         * Split the string into arguments.
         */
        parse(buf, args);

        /*
         * Execute the command.
         */
        execute(args);
    }
}

/*
 * parse--split the command in buf into
 *         individual arguments.
 */
parse(buf, args)
char *buf;
char **args;
{
    while (*buf != NULL) {
        /*
         * Strip whitespace.  Use nulls, so
         * that the previous argument is terminated
         * automatically.
         */
        while ((*buf == ' ') || (*buf == '\t'))
            *buf++ = NULL;

        /*
         * Save the argument.
         */
        *args++ = buf;

        /*
         * Skip over the argument.
         */
        while ((*buf != NULL) && (*buf != ' ') && (*buf != '\t'))
            buf++;
    }

    *args = NULL;
}

/*
 * execute--spawn a child process and execute
 *           the program.
 */
execute(args)
char **args;
{
    int pid, status;

    /*
     * Get a child process.
     */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);

/* NOTE: perror() produces a short  error  message  on  the  standard
          error describing the last error encountered during a call to
          a system or library function.
      */
    }

    /*
     * The child executes the code inside the if.
     */
    if (pid == 0) {
        execvp(*args, args);
        perror(*args);
        exit(1);

       /* NOTE: The execv() vnd execvp versions of execl() are useful when the
          number  of  arguments is unknown in advance;
          The arguments to execv() and execvp()  are the name
          of the file to be executed and a vector of strings  contain-
          ing  the  arguments.   The last argument string must be fol-
          lowed by a 0 pointer.

          execlp() and execvp() are called with the same arguments  as
          execl()  and  execv(),  but duplicate the shell's actions in
          searching for an executable file in a list  of  directories.
          The directory list is obtained from the environment.
        */
    }

    /*
     * The parent executes the wait.
     */
    while (wait(&status) != pid)
        /* empty */ ;
}
```

21.6 Exerises

Exercise 21.1 *Use* `popen()` *to pipe the* `rwho` *(UNIX command) output into* `more` *(UNIX command) in a C program.*

Chapter 22

Interprocess Communication (IPC), Pipes

We have now began to see how multiple processes may be running on a machine and maybe be controlled (spawned by `fork()` by one of our programs.

In numerous applications there is clearly a need for these processes to communicate with each exchanging data or control information. There are a few methods which can accomplish this task. We will consider:

- Pipes
- Signals
- Message Queues
- Semaphores
- Shared Memory
- Sockets

In this chapter, we will study the piping of two processes. We will study the others in turn in subsequent chapters.

22.1 Piping in a C program: <`stdio.h`>

Piping is a process where the input of one process is made the input of another. We have seen examples of this from the UNIX command line using |.

We will now see how we do this from C programs.

We will have two (or more) `forked` processes and will communicate between them.

We must first open a *pipe*

UNIX allows two ways of opening a pipe.

22.2 popen() — Formatted Piping

`FILE *popen(char *command, char *type)` — opens a pipe for I/O where the command is the process that will be connected to the calling process thus creating the *pipe*. The type is either "r" – for reading, or "w" for writing.

`popen()` returns is a stream pointer or NULL for any errors.

A pipe opened by `popen()` should always be closed by `pclose(FILE *stream)`.

We use `fprintf()` and `fscanf()` to communicate with the pipe's `stream`.

22.3 pipe() — Low level Piping

`int pipe(int fd[2])` — creates a pipe and returns two file descriptors, `fd[0]`, `fd[1]`. `fd[0]` is opened for reading, `fd[1]` for writing.

`pipe()` returns 0 on success, -1 on failure and sets `errno` accordingly.

The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using `read()` and `write()`.

Pipes opened with `pipe()` should be closed with `close(int fd)`.

Example: Parent writes to a child

```
int pdes[2];

pipe(pdes);
if ( fork() == 0 )
  { /* child */
```

```
    close(pdes[1]); /* not required */
    read( pdes[0]); /* read from parent */
    .....
  }
else
  { close(pdes[0]); /* not required */
    write( pdes[1]); /* write to child */
    .....
  }
```

An futher example of piping in a C program is `plot.c` and subroutines and it performs as follows:

- The program has two modules `plot.c` (main) and `plotter.c`.
- The program relies on you having installed the freely *gnuplot* graph drawing program in the directory `/usr/local/bin/` (in the listing below at least) — this path could easily be changed.
- The program `plot.c` calls *gnuplot*
- Two Data Stream is generated from Plot
  - $y = sin(x)$
  - $y = sin(1/x)$
- 2 Pipes created — 1 per Data Stream.
- $^\circ$ Gnuplot produces "live" drawing of output.

The code listing for `plot.c` is:

```
/* plot.c - example of unix pipe. Calls gnuplot graph drawing package to draw
   graphs from within a C program. Info is piped to gnuplot */
/* Creates 2 pipes one will draw graphs of y=0.5 and y = random 0-1.0 */
/* the other graphs of y = sin (1/x) and y = sin x */

/* Also user a plotter.c module */
/* compile: cc -o plot plot.c plotter.c */
```

```
#include "externals.h"
#include <signal.h>

#define DEG_TO_RAD(x) (x*180/M_PI)

double drand48();
void quit();

FILE *fp1, *fp2, *fp3, *fp4, *fopen();

main()
{   float i;
    float y1,y2,y3,y4;

    /* open files which will store plot data */
    if ( ((fp1 = fopen("plot11.dat","w")) == NULL) ||
           ((fp2 = fopen("plot12.dat","w")) == NULL) ||
            ((fp3 = fopen("plot21.dat","w")) == NULL) ||
             ((fp4 = fopen("plot22.dat","w")) == NULL) )
              { printf("Error can't open one or more data files\n");
                exit(1);
              }

    signal(SIGINT,quit); /* trap ctrl-c call quit fn */
    StartPlot();
    y1 = 0.5;
    srand48(1); /* set seed */
    for (i=0;;i+=0.01) /* increment i forever use ctrl-c to quit prog */
      { y2 =  (float) drand48();
        if (i == 0.0)
           y3 = 0.0;
       else
           y3 = sin(DEG_TO_RAD(1.0/i));
        y4 = sin(DEG_TO_RAD(i));

        /* load files */
        fprintf(fp1,"%f %f\n",i,y1);
        fprintf(fp2,"%f %f\n",i,y2);
        fprintf(fp3,"%f %f\n",i,y3);
        fprintf(fp4,"%f %f\n",i,y4);

        /* make sure buffers flushed so that gnuplot */
        /*  reads up to data file */
        fflush(fp1);
        fflush(fp2);
        fflush(fp3);
        fflush(fp4);

        /* plot graph */
        PlotOne();
        usleep(250); /* sleep for short time */
      }
}

void quit()
{  printf("\nctrl-c caught:\n Shutting down pipes\n");
   StopPlot();

   printf("closing data files\n");
   fclose(fp1);
   fclose(fp2);
   fclose(fp3);
   fclose(fp4);

   printf("deleting data files\n");
   RemoveDat();
}
```

The plotter.c module is as follows:

```
/* plotter.c module */
/* contains routines to plot a data file produced by another program  */
/* 2d data plotted in this version                                    */
/**********************************************************************/
```

```
#include "externals.h"

static FILE *plot1,
       *plot2,
       *ashell;

static char *startplot1 = "plot [] [0:1.1]'plot11.dat' with lines,
            'plot12.dat' with lines\n";

static char *startplot2 = "plot 'plot21.dat' with lines,
            'plot22.dat' with lines\n";

static char *replot = "replot\n";
static char *command1= "/usr/local/bin/gnuplot> dump1";
static char *command2= "/usr/local/bin/gnuplot> dump2";
static char *deletefiles = "rm plot11.dat plot12.dat plot21.dat plot22.dat"
static char *set_term = "set terminal x11\n";

void
StartPlot(void)
 { plot1 = popen(command1, "w");
   fprintf(plot1, "%s", set_term);
   fflush(plot1);
   if (plot1 == NULL)
      exit(2);
   plot2 = popen(command2, "w");
   fprintf(plot2, "%s", set_term);
   fflush(plot2);
   if (plot2 == NULL)
      exit(2);
 }

void
RemoveDat(void)
 { ashell = popen(deletefiles, "w");
   exit(0);
 }
```

```
void
StopPlot(void)
 { pclose(plot1);
   pclose(plot2);
 }

void
PlotOne(void)
 { fprintf(plot1, "%s", startplot1);
   fflush(plot1);

   fprintf(plot2, "%s", startplot2);
   fflush(plot2);
 }

void
RePlot(void)
 { fprintf(plot1, "%s", replot);
   fflush(plot1);
 }
```

The header file `externals.h` contains the following:

```
/* externals.h */
#ifndef EXTERNALS
#define EXTERNALS

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* prototypes */

void StartPlot(void);
void RemoveDat(void);
void StopPlot(void);
void PlotOne(void);
void RePlot(void);
#endif
```

22.4 Exercises

Exercise 22.1 *Setup a two-way pipe between parent and child processes in a C program. i.e. both can send and receive signals.*

Chapter 23

IPC:Interrupts and Signals: `<signal.h>`

In this section will look at ways in which two processes can communicate. When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

Signals are software generated interrupts that are sent to a process when a event happens. Signals can be synchronously generated by an error in an application, such as `SIGFPE` and `SIGSEGV`, but most signals are asynchronous. Signals can be posted to a process when the system detects a software event, such as a user entering an interrupt or stop or a kill request from another process. Signals can also be come directly from the OS kernel when a hardware event such as a bus error or an illegal instruction is encountered. The system defines a set of signals that can be posted to a process. Signal delivery is analogous to hardware interrupts in that a signal can be blocked from being delivered in the future. Most signals cause termination of the receiving process if no action is taken by the process in response to the signal. Some signals stop the receiving process and other signals can be ignored. Each signal has a default action which is one of the following:

- The signal is discarded after being received
- The process is terminated after the signal is received
- A core file is written, then the process is terminated

- Stop the process after the signal is received

Each signal defined by the system falls into one of five classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Process control
- Resource control

Macros are defined in <`signal.h`> header file for common signals.

These include:

SIGHUP 1 /* hangup */	SIGINT 2 /* interrupt */
SIGQUIT 3 /* quit */	SIGILL 4 /* illegal instruction *
SIGABRT 6 /* used by abort */	SIGKILL 9 /* hard kill */
SIGALRM 14 /* alarm clock */	
SIGCONT 19 /* continue a stopped process */	
SIGCHLD 20 /* to parent on child stop or exit */	

Signals can be numbered from 0 to 31.

23.1 Sending Signals — `kill()`, `raise()`

There are two common functions used to send signals

`int kill(int pid, int signal)` – a system call that send a `signal` to a process, `pid`. If pid is greater than zero, the signal is sent to the process whose process ID is equal to pid. If pid is 0, the signal is sent to all processes, except system processes.

`kill()` returns 0 for a successful call, -1 otherwise and sets `errno` accordingly.

`int raise(int sig)` sends the signal sig to the executing program. `raise()` actually uses `kill()` to send the signal to the executing program:

```
kill(getpid(), sig);
```

There is also a UNIX command called kill that can be used to send signals from the command line – see `man` pages.

NOTE: that unless caught or ignored, the `kill` signal terminates the process. Therefore protection is built into the system.

Only processes with certain access privileges can be killed off.

Basic rule: *only processes that have the same user can send/receive messages.*

The `SIGKILL` signal cannot be caught or ignored and will always terminate a process.

For example`kill(getpid(),SIGINT);` would send the interrupt signal to the id of the calling process.

This would have a similar effect to `exit()` command. Also `ctrl-c` typed from the command sends a `SIGINT` to the process currently being.

`unsigned int alarm(unsigned int seconds)` — sends the signal `SIGALRM` to the invoking process after seconds seconds.

23.2 Signal Handling — `signal()`

An application program can specify a function called a signal handler to be invoked when a specific signal is received. When a signal handler is invoked on receipt of a signal, it is said to catch the signal. A process can deal with a signal in one of the following ways:

- The process can let the default action happen
- The process can block the signal (some signals cannot be ignored)
- the process can catch the signal with a handler.

Signal handlers usually execute on the current stack of the process. This lets the signal handler return to the point that execution was interrupted in the process. This can be changed on a per-signal basis so that a signal handler executes on a special stack. If a process must resume in a different context than the interrupted one, it must restore the previous context itself

Receiving signals is straighforward with the function:

`int (*signal(int sig, void (*func)()))()` — that is to say the function `signal()` will call the `func` functions if the process receives a signal `sig`. Signal returns a pointer to function `func` if successful or it returns an error to `errno` and -1 otherwise.

`func()` can have three values:

`SIG_DFL` — a pointer to a system default function `SID_DFL()`, which will terminate the process upon receipt of `sig`.

`SIG_IGN` — a pointer to system ignore function `SIG_IGN()` which will disregard the `sig` action (UNLESS it is `SIGKILL`).

A function address — a user specified function.

`SIG_DFL` and `SIG_IGN` are defined in `signal.h` (standard library) header file.

Thus to ignore a `ctrl-c` command from the command line. we could do:

```
signal(SIGINT, SIG_IGN);
```

TO reset system so that `SIGINT` causes a termination at any place in our program, we would do:

```
signal(SIGINT, SIG_DFL);
```

So lets write a program to trap a `ctrl-c` but not quit on this signal. We have a function `sigproc()` that is executed when we trap a `ctrl-c`. We will also set another function to quit the program if it traps the `SIGQUIT` signal so we can terminate our program:

```
#include <stdio.h>

void sigproc(void);

void quitproc(void);

main()
{ signal(SIGINT, sigproc);
  signal(SIGQUIT, quitproc);
  printf(``ctrl-c disabled use ctrl-\\ to quit \n'');
  for(;;); /* infinite loop */ }

void sigproc()
{ signal(SIGINT, sigproc); /* */
  /* NOTE some versions of UNIX will reset signal to default
  after each call.  So for portability reset signal each time */

  printf(``you have pressed ctrl-c \n'');
}

void quitproc()
{ printf(``ctrl-\\ pressed to quit\n'');
  exit(0); /* normal exit status */
}
```

23.3 sig_talk.c — complete example program

Let us now write a program that communicates between child and parent processes using `kill()` and `signal()`.

`fork()` creates the child process from the parent. The `pid` can be checked to decide whether it is the child (== 0) or the parent (pid = child process id).

The parent can then send messages to child using the pid and `kill()`.

The child picks up these signals with `signal()` and calls appropriate functions.

An example of communicating process using signals is `sig_talk.c`:

```
/* sig_talk.c --- Example of how 2 processes can talk */
/* to each other using kill() and signal() */
/* We will fork() 2 process and let the parent send a few */
/* signals to it`s child  */
/* cc sig_talk.c -o sig_talk  */

#include <stdio.h>
#include <signal.h>

void sighup(); /* routines child will call upon sigtrap */
void sigint();
void sigquit();

main()
{ int pid;

  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

   if (pid == 0)
     { /* child */
       signal(SIGHUP,sighup); /* set function calls */
       signal(SIGINT,sigint);
       signal(SIGQUIT, sigquit);
       for(;;); /* loop for ever */
     }
  else /* parent */
     {  /* pid hold id of child */
       printf("\nPARENT: sending SIGHUP\n\n");
       kill(pid,SIGHUP);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGINT\n\n");
       kill(pid,SIGINT);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGQUIT\n\n");
       kill(pid,SIGQUIT);
       sleep(3);
     }
}

void sighup()

{  signal(SIGHUP,sighup); /* reset signal */
   printf("CHILD: I have received a SIGHUP\n");
}

void sigint()

{  signal(SIGINT,sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

void sigquit()

{ printf("My DADDY has Killed me!!!\n");
  exit(0);
}
```

23.4 Other signal functions

There are a few other functions defined in `signal.h`:

`int sighold(int sig)` — adds `sig` to the calling process's signal mask

`int sigrelse(int sig)` — removes `sig` from the calling process's signal mask

`int sigignore(int sig)` — sets the disposition of `sig` to `SIG_IGN`

`int sigpause(int sig)` — removes `sig` from the calling process's signal mask and suspends the calling process until a signal is received

Chapter 24

IPC:Message Queues:<sys/msg.h>

The basic idea of a *message queue* is a simple one.

Two (or more) processes can exchange information via access to a common system message queue. The *sending* process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure 24.1). Each message is given an identification or `type` so that processes can select the appropriate message. Process must share a common `key` in order to gain access to the queue in the first place (subject to other permissions — see below).

IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the `msgget` function see below) Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively.

When a message is sent, its text is copied to the message queue. The `msgsnd()` and `msgrcv()` functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer — the process is not suspended as a result of sending or

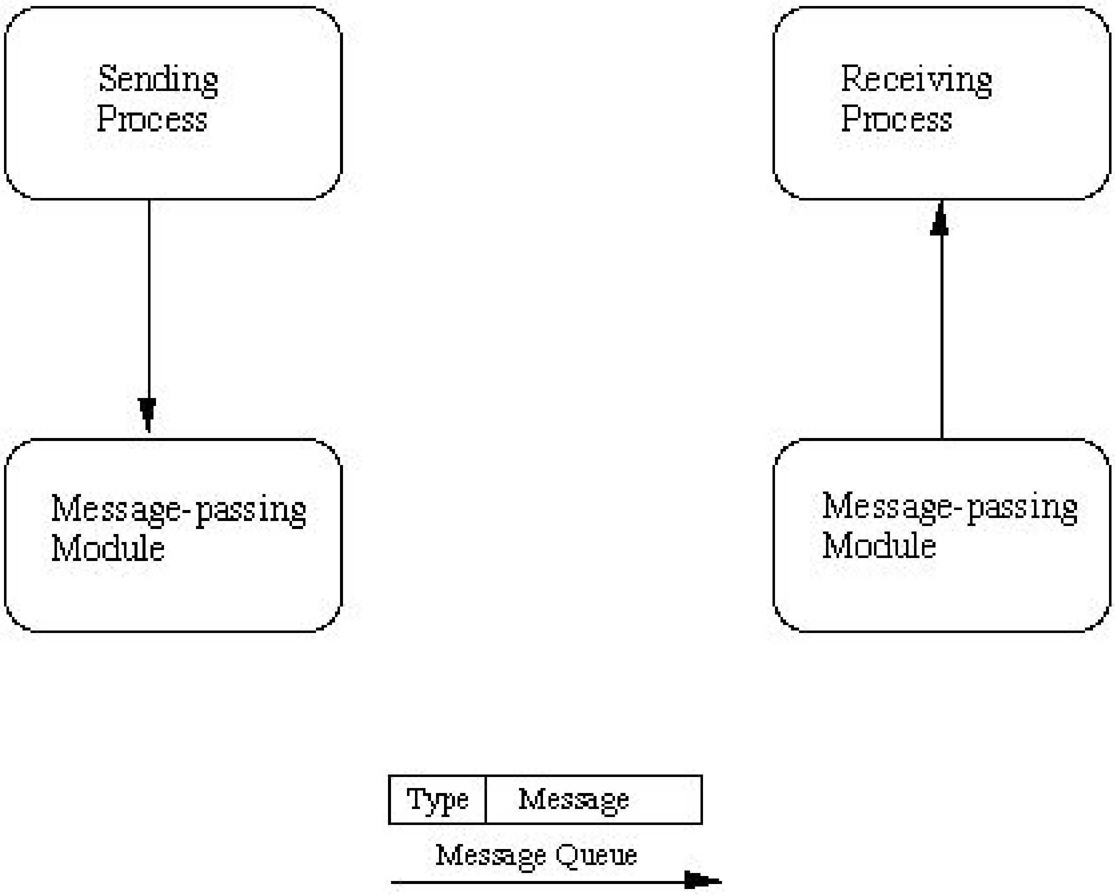


Figure 24.1: Basic Message Passing

receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

24.1 Initialising the Message Queue

The `msgget()` function initializes a new message queue:

```
int msgget(key_t key, int msgflg)
```

It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the `msgget()` function.

```
#include <sys/ipc.h>;
#include <sys/msg.h>;

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
    perror("msgget: msgget failed");
    exit(1);
   } else
    (void) fprintf(stderr, &ldquo;msgget succeeded");
...
```

24.2 IPC Functions, Key Arguments, and Creation Flags: <sys/ipc.h>

Processes requesting access to an IPC facility must be able to identify it. To do this, functions that initialize or provide access to an IPC facility use a `key_t key` argument. (`key_t` is essentially an `int` type defined in <sys/types.h>

The `key` is an arbitrary value or one that can be derived from a common seed at run time. One way is with `ftok()` , which converts a filename to a key value that is unique within the system. Functions that initialize or get access to messages (also semaphores or shared memory see later) return an ID number of type int. IPC functions that perform read, write, and control operations use this ID. If the key argument is specified as `IPC_PRIVATE`, the call initializes a new instance of an IPC facility that is private to the creating process. When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the function tries to create the facility if it does not exist already. When called with both the `IPC_CREAT` and `IPC_EXCL` flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with `IPC_EXCL` in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If `IPC_CREAT` is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/tmp",
key), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a `key` based on the string ("/tmp"). The second argument evaluates to the combined permissions and control flags.

24.3 Controlling message queues

The `msgctl()` function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using `msgctl()` Also, any process with permission to do so can use `msgctl()` for control operations.

The `msgctl()` function is prototypes as follows:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf )
```

The `msqid` argument must be the ID of an existing message queue. The `cmd argument` is one of:

`IPC_STAT` — Place information about the status of the queue in the data structure pointed to by `buf`. The process must have read permission for this call to succeed.

`IPC_SET` — Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

`IPC_RMID` — Remove the message queue specified by the `msqid` argument.

The following code illustrates the `msgctl()` function with all its various flags:

```
#include<sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
if (msgctl(msqid, IPC_STAT, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
```

```
if (msgctl(msqid, IPC_SET, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
```

24.4 Sending and Receiving Messages

The `msgsnd()` and `msgrcv()` functions send and receive messages, respectively:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz,
          int msgflg);

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
          int msgflg);
```

The `msqid` argument **must** be the ID of an existing message queue. The `msgp` argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
 struct mymsg {
      long      mtype;    /* message type */
      char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

The `msgsz` argument specifies the length of the message in bytes.

The structure member `msgtype` is the received message's type as specified by the sending process.

The argument `msgflg` specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to `msg_qbytes`.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (`msgflg & IPC_NOWAIT`) is non-zero, the message will not be sent and the calling process will return immediately.
- If (`msgflg & IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue identifier `msqid` is removed from the system; when this occurs, `errno` is set equal to `EIDRM` and -1 is returned.
  - The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

  Upon successful completion, the following actions are taken with respect to the data structure associated with `msqid`:

  - `msg_qnum` is incremented by 1.
  - `msg_lspid` is set equal to the process ID of the calling process.
  - `msg_stime` is set equal to the current time.

The following code illustrates `msgsnd()` and `msgrcv()`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

...

int msgflg; /* message flags for the operation */
struct msgbuf *msgp; /* pointer to the message buffer */
int msgsz; /* message size */
long msgtyp; /* desired message type */
int msqid /* message queue ID to be used */

...

msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
- sizeof msgp->mtext + maxmsgsz));

if (msgp == NULL) {
(void) fprintf(stderr, "msgop: %s %d byte messages.\n",
"could not allocate message buffer for", maxmsgsz);
exit(1);

...

msgsz = ...
msgflg = ...

if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
perror("msgop: msgsnd failed");
...
msgsz = ...
msgtyp = first_on_queue;
msgflg = ...
if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
perror("msgop: msgrcv failed");
...
```

24.5 POSIX Messages: <mqueue.h>

The POSIX message queue functions are:

`mq_open()` — Connects to, and optionally creates, a named message queue.

`mq_close()` — Ends the connection to an open message queue.

`mq_unlink()` — Ends the connection to an open message queue and causes the queue to be removed when the last process closes it.

`mq_send()` — Places a message in the queue.

`mq_receive()` — Receives (removes) the oldest, highest priority message from the queue.

`mq_notify()` — Notifies a process or thread that a message is available in the queue.

`mq_setattr()` — Set or get message queue attributes.

The basic operation of these functions is as described above. For full function prototypes and further information see the UNIX `man pages`

24.6 Example: Sending messages between two processes

The following two programs should be compiled and run *at the same time* to illustrate basic principle of message passing:

`message_send.c` — Creates a message queue and sends one message to the queue.

`message_rec.c` — Reads the message from the queue.

24.6.1 `message_send.c` — creating and sending to a simple message queue

The full code listing for `message_send.c` is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>

#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
         long    mtype;
         char    mtext[MSGSZ];
         } message_buf;
```

```
main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    message_buf sbuf;
    size_t buf_length;

    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\
%#o)\n",
key, msgflg);

    if ((msqid = msgget(key, msgflg )) < 0) {
        perror("msgget");
        exit(1);
    }
    else
     (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid)


    /*
     * We'll send message type 1
     */

    sbuf.mtype = 1;

    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    (void) strcpy(sbuf.mtext, "Did you get this?");

    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
    buf_length = strlen(sbuf.mtext) ;



    /*
     * Send a message.
     */
    if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
       printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
        perror("msgsnd");
        exit(1);
    }

   else
      printf("Message: \"%s\" Sent\n", sbuf.mtext);

    exit(0);
}
```

The essential points to note here are:

- The Message queue is created with a basic `key` and message flag `msgflg = IPC_CREAT | 0666` — create queue and make it read and appendable by all.
- A message of type (`sbuf.mtype`) 1 is sent to the queue with the message "`Did you get this?`"

24.6.2 `message_rec.c` — receiving the above message

The full code listing for `message_send.c`'s companion process, `message_rec.c` is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
```

```
#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
} message_buf;


main()
{
    int msqid;
    key_t key;
    message_buf  rbuf;

    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

    if ((msqid = msgget(key, 0666)) < 0) {
        perror("msgget");
        exit(1);
    }


    /*
     * Receive an answer of message type 1.
     */
    if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
        perror("msgrcv");
        exit(1);
    }

    /*
     * Print the answer.
     */
    printf("%s\n", rbuf.mtext);
    exit(0);
}
```

The essential points to note here are:

- The Message queue is opened with `msgget` (message flag `0666`) and the *same* `key` as `message_send.c`.
- A message of the *same* type 1 is received from the queue with the message "`Did you get this?`" stored in `rbuf.mtext`.

24.7 Some further example message queue programs

The following suite of programs can be used to investigate interactively a variety of massage passing ideas (see exercises below).

The message queue **must** be initialised with the `msgget.c` program. The effects of controlling the queue and sending and receiving messages can be investigated with `msgctl.c` and `msgop.c` respectively.

24.7.1 `msgget.c`: Simple Program to illustrate `msget()`

```
/*
 * msgget.c: Illustrate the msgget() function.
 * This is a simple exerciser of the msgget() function. It prompts
 * for the arguments, makes the call, and reports the results.
 */

#include <stdio.h>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

extern void  exit();
extern void  perror();

main()
{
 key_t  key;  /* key to be passed to msgget() */
 int  msgflg,  /* msgflg to be passed to msgget() */
   msqid;  /* return value from msgget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);
 (void) fprintf(stderr, "\nExpected flags for msgflg argument
are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter msgflg value: ");
 (void) scanf("%i", &msgflg);

 (void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,
%#o)\n",
 key, msgflg);
 if ((msqid = msgget(key, msgflg)) == -1)
 {
  perror("msgget: msgget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
   "msgget: msgget succeeded: msqid = %d\n", msqid);
  exit(0);
 }
}
```

24.7.2 `msgctl.c`Sample Program to Illustrate `msgctl()`

```
/*
 * msgctl.c:  Illustrate the msgctl() function.
 *
 * This is a simple exerciser of the msgctl() function.  It allows
 * you to perform one control operation on one message queue.  It
 * gives up immediately if any control operation fails, so be
careful
 * not to set permissions to preclude read permission; you won't
be
 * able to reset the permissions with this code if you do.
 */
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>
#include   <time.h>

static void do_msgctl();
extern void exit();
extern void perror();
static char warning_message[] = "If you remove read permission
for \
     yourself, this program will fail frequently!";
```

```
main()
{
 struct msqid_ds buf;     /* queue descriptor buffer for IPC_STAT
          and IP_SET commands */
 int    cmd,  /* command to be given to msgctl() */
     msqid;  /* queue ID to be given to msgctl() */

 (void fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the msqid and cmd arguments for the msgctl() call. */
 (void) fprintf(stderr,
  "Please enter arguments for msgctls() as requested.");
 (void) fprintf(stderr, "\nEnter the msqid: ");
 (void) scanf("%i", &msqid);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\nEnter the value for the command: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_SET:
   /* Modify settings in the message queue control structure.
*/
   (void) fprintf(stderr, "Before IPC_SET, get current
values:");
   /* fall through to IPC_STAT processing */
  case IPC_STAT:
   /* Get a copy of the current message queue control
    * structure and show it to the user. */
   do_msgctl(msqid, IPC_STAT, &buf);
   (void) fprintf(stderr, ]
   "msg_perm.uid = %d\n", buf.msg_perm.uid);
   (void) fprintf(stderr,
   "msg_perm.gid = %d\n", buf.msg_perm.gid);
   (void) fprintf(stderr,
   "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
   (void) fprintf(stderr,
   "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
   (void) fprintf(stderr, "msg_perm.mode = %#o, ",
   buf.msg_perm.mode);
   (void) fprintf(stderr, "access permissions = %#o\n",
   buf.msg_perm.mode & 0777);
   (void) fprintf(stderr, "msg_cbytes = %d\n",
       buf.msg_cbytes);
   (void) fprintf(stderr, "msg_qbytes = %d\n",
       buf.msg_qbytes);
   (void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
   (void) fprintf(stderr, "msg_lspid = %d\n",
       buf.msg_lspid);
   (void) fprintf(stderr, "msg_lrpid = %d\n",
       buf.msg_lrpid);
   (void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ?
   ctime(&buf.msg_stime) : "Not Set\n");
   (void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ?
   ctime(&buf.msg_rtime) : "Not Set\n");
   (void) fprintf(stderr, "msg_ctime = %s",
       ctime(&buf.msg_ctime));
   if (cmd == IPC_STAT)
    break;
   /*  Now continue with IPC_SET. */
   (void) fprintf(stderr, "Enter msg_perm.uid: ");
   (void) scanf ("%hi", &buf.msg_perm.uid);
   (void) fprintf(stderr, "Enter msg_perm.gid: ");
   (void) scanf("%hi", &buf.msg_perm.gid);
   (void) fprintf(stderr, "%s\n", warning_message);
   (void) fprintf(stderr, "Enter msg_perm.mode: ");
   (void) scanf("%hi", &buf.msg_perm.mode);
   (void) fprintf(stderr, "Enter msg_qbytes: ");
   (void) scanf("%hi", &buf.msg_qbytes);
```

```
   do_msgctl(msqid, IPC_SET, &buf);
   break;
  case IPC_RMID:
  default:
   /* Remove the message queue or try an unknown command. */
   do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
   break;
 }
 exit(0);
}

/*
 * Print indication of arguments being passed to msgctl(), call
 * msgctl(), and report the results. If msgctl() fails, do not
 * return; this example doesn't deal with errors, it just reports
 * them.
 */
static void
do_msgctl(msqid, cmd, buf)
struct msqid_ds    *buf;  /* pointer to queue descriptor buffer */
int    cmd,  /* command code */
    msqid;  /* queue ID */
{
 register int rtrn;  /* hold area for return value from msgctl()
*/

 (void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d,
%s)\n",
   msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
 rtrn = msgctl(msqid, cmd, buf);
 if (rtrn == -1) {
  perror("msgctl: msgctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "msgctl: msgctl returned %d\n",
      rtrn);
 }
}
```

24.7.3 msgop.c: Sample Program to Illustrate msgsnd() and msgrcv()

```
/*
 * msgop.c: Illustrate the msgsnd() and msgrcv() functions.
 *
 * This is a simple exerciser of the message send and receive
 * routines. It allows the user to attempt to send and receive as
many
 * messages as wanted to or from one message queue.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

static int ask();
extern void exit();
extern char *malloc();
extern void perror();

char first_on_queue[] = "-> first message on queue",
 full_buf[] = "Message buffer overflow. Extra message text\
      discarded.";

main()
{
 register int    c;  /* message text input */
 int    choice; /* user's selected operation code */
 register int    i;  /* loop control for mtext */
 int    msgflg; /* message flags for the operation */
 struct msgbuf     *msgp;  /* pointer to the message buffer */
 int    msgsz;  /* message size */
 long    msgtyp;  /* desired message type */
 int    msqid,  /* message queue ID to be used */
     maxmsgsz, /* size of allocated message buffer */
     rtrn;  /* return value from msgrcv or msgsnd */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* Get the message queue ID and set up the message buffer. */
 (void) fprintf(stderr, "Enter msqid: ");
 (void) scanf("%i", &msqid);
 /*
  * Note that <sys/msg.h> includes a definition of struct
msgbuf
  * with the mtext field defined as:
  *   char mtext[1];
  * therefore, this definition is only a template, not a
structure
  * definition that you can use directly, unless you want only
to
  * send and receive messages of 0 or 1 byte. To handle this,
  * malloc an area big enough to contain the template - the size
  * of the mtext template field + the size of the mtext field
  * wanted. Then you can use the pointer returned by malloc as a
  * struct msgbuf with an mtext field of the size you want. Note
  * also that sizeof msgp->mtext is valid even though msgp
isn't
  * pointing to anything yet. Sizeof doesn't dereference msgp,
but
  * uses its type to figure out what you are asking about.
  */
 (void) fprintf(stderr,
  "Enter the message buffer size you want:");
 (void) scanf("%i", &maxmsgsz);
 if (maxmsgsz < 0) {
  (void) fprintf(stderr, "msgop: %s\n",
    "The message buffer size must be >= 0.");
  exit(1);
 }
 msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct
msgbuf)
    - sizeof msgp->mtext + maxmsgsz));
 if (msgp == NULL) {
  (void) fprintf(stderr, "msgop: %s %d byte messages.\n",
    "could not allocate message buffer for", maxmsgsz);
  exit(1);
 }
 /* Loop through message operations until the user is ready to
  quit. */
 while (choice = ask()) {
  switch (choice) {
  case 1: /* msgsnd() requested: Get the arguments, make the
    call, and report the results. */
   (void) fprintf(stderr, "Valid msgsnd message %s\n",
    "types are positive integers.");
   (void) fprintf(stderr, "Enter msgp->mtype: ");
   (void) scanf("%li", &msgp->mtype);
   if (maxmsgsz) {
    /* Since you've been using scanf, you need the loop
       below to throw away the rest of the input on the
       line after the entered mtype before you start
       reading the mtext. */
    while ((c = getchar()) != '\n' && c != EOF);
    (void) fprintf(stderr, "Enter a %s:\n",
        "one line message");
    for (i = 0; ((c = getchar()) != '\n'); i++) {
     if (i >= maxmsgsz) {
      (void) fprintf(stderr, "\n%s\n", full_buf);
      while ((c = getchar()) != '\n');
      break;
     }
     msgp->mtext[i] = c;
    }
    msgsz = i;
   } else
```

```
    msgsz = 0;
   (void) fprintf(stderr,"\nMeaningful msgsnd flag is:\n");
   (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
    IPC_NOWAIT);
   (void) fprintf(stderr, "Enter msgflg: ");
   (void) scanf("%i", &msgflg);
   (void) fprintf(stderr, "%s(%d, msgp, %d, %#o)\n",
    "msgop: Calling msgsnd", msqid, msgsz, msgflg);
   (void) fprintf(stderr, "msgp->mtype = %ld\n",
       msgp->mtype);
   (void) fprintf(stderr, "msgp->mtext = \"");
   for (i = 0; i < msgsz; i++)
    (void) fputc(msgp->mtext[i], stderr);
    (void) fprintf(stderr, "\"\n");
    rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
    if (rtrn == -1)
     perror("msgop: msgsnd failed");
    else
     (void) fprintf(stderr,
        "msgop: msgsnd returned %d\n", rtrn);
    break;
  case 2: /* msgrcv() requested: Get the arguments, make the
       call, and report the results. */
   for (msgsz = -1; msgsz < 0 || msgsz > maxmsgsz;
      (void) scanf("%i", &msgsz))
    (void) fprintf(stderr, "%s (0 <= msgsz <= %d): ",
        "Enter msgsz", maxmsgsz);
   (void) fprintf(stderr, "msgtyp meanings:\n");
   (void) fprintf(stderr, "\t 0 %s\n", first_on_queue);
   (void) fprintf(stderr, "\t>0 %s of given type\n",
    first_on_queue);
   (void) fprintf(stderr, "\t<0 %s with type <= |msgtyp|\n",
       first_on_queue);
   (void) fprintf(stderr, "Enter msgtyp: ");
   (void) scanf("%li", &msgtyp);
   (void) fprintf(stderr,
       "Meaningful msgrcv flags are:\n");
   (void) fprintf(stderr, "\tMSG_NOERROR =\t%#8.8o\n",
       MSG_NOERROR);
   (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
       IPC_NOWAIT);
   (void) fprintf(stderr, "Enter msgflg: ");
   (void) scanf("%i", &msgflg);
   (void) fprintf(stderr, "%s(%d, msgp, %d, %ld, %#o);\n",
       "msgop: Calling msgrcv", msqid, msgsz,
       msgtyp, msgflg);
   rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
   if (rtrn == -1)
    perror("msgop: msgrcv failed");
   else {
    (void) fprintf(stderr, "msgop: %s %d\n",
        "msgrcv returned", rtrn);
    (void) fprintf(stderr, "msgp->mtype = %ld\n",
        msgp->mtype);
    (void) fprintf(stderr, "msgp->mtext is: \"");
    for (i = 0; i < rtrn; i++)
     (void) fputc(msgp->mtext[i], stderr);
    (void) fprintf(stderr, "\"\n");
   }
   break;
  default:
   (void) fprintf(stderr, "msgop: operation unknown\n");
   break;
  }
 }
 exit(0);
}

/*
 * Ask the user what to do next. Return the user's choice code.
 * Don't return until the user selects a valid choice.
 */
static
ask()
{
 int response; /* User's response. */
 do {
  (void) fprintf(stderr, "Your options are:\n");
  (void) fprintf(stderr, "\tExit =\t0 or Control-D\n");
  (void) fprintf(stderr, "\tmsgsnd =\t1\n");
  (void) fprintf(stderr, "\tmsgrcv =\t2\n");
  (void) fprintf(stderr, "Enter your choice: ");

  /* Preset response so "^D" will be interpreted as exit. */
  response = 0;
  (void) scanf("%i", &response);
 } while (response < 0 || response > 2);

 return(response);
}
```

24.8 Exercises

Exercise 24.1 *Write a 2 programs that will both send and messages and construct the following dialog between them*

- *(Process 1) Sends the message "Are you hearing me?"*
- *(Process 2) Receives the message and replies "Loud and Clear".*
- *(Process 1) Receives the reply and then says "I can hear you too".*

Exercise 24.2 *Compile the programs* `msgget.c`*,* `msgctl.c` *and* `msgop.c` *and then*

- *investigate and understand fully the operations of the flags (access, creation* etc. *permissions) you can set interactively in the programs.*
- *Use the programs to:*
  - *Send and receive messages of two different message* `type`*s.*

  - *Place several messages on the queue and inquire about the state of the queue with* `msgctl.c`*. Add/delete a few messages (using* `msgop.c` *and perform the inquiry once more.*
  - *Use* `msgctl.c` *to alter a message on the queue.*
  - *Use* `msgctl.c` *to delete a message from the queue.*

Exercise 24.3 *Write a* server *program and two* client *programs so that the* server *can communicate privately to* each client *individually via a* single *message queue.*

Exercise 24.4 *Implement a* blocked *or* synchronous *method of message passing using signal interrupts.*

Chapter 25

IPC:Semaphores

Semaphores are a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads: consider a stretch of railroad in which there is a single track over which only one train at a time is allowed. Guarding this track is a semaphore. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter. In the computer version, a semaphore appears to be a simple integer. A process (or a thread) waits for permission to proceed by waiting for the integer to become 0. The signal if it proceeds signals that this by performing incrementing the integer by 1. When it is finished, the process changes the semaphore's value by subtracting one from it.

Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments.

Semaphores can be operated on as individual units or as elements in a set. Because System V IPC semaphores can be in a large array, they are extremely heavy weight. Much lighter weight semaphores are available in the threads library (see `man semaphore` and also Chapter 30.3) and POSIX semaphores (see below briefly). Threads library semaphores must be used with mapped memory . A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements.

In a similar fashion to message queues, the semaphore set must be initialized using `semget()`; the semaphore creator can change its ownership or permissions using `semctl()`; and semaphore operations are performed via the `semop()` function. These are now discussed below:

25.1 Initializing a Semaphore Set

The function `semget()` initializes or gains access to a semaphore. It is prototyped by:

```
int semget(key_t key, int nsems, int semflg);
```

When the call succeeds, it returns the semaphore ID (`semid`).

The `key` argument is a access value associated with the semaphore ID.

The `nsems` argument specifies the number of elements in a semaphore array. The call fails when `nsems` is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument ensures that it will succeed.

The `semflg` argument specifies the initial access permissions and creation control flags.

The following code illustrates the semget() function.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

...
key_t key; /* key to pass to semget() */
int semflg; /* semflg to pass tosemget() */
int nsems; /* nsems to pass to semget() */
int semid; /* return value from semget() */

...

key = ...
nsems = ...
semflg = ... ...
if ((semid = semget(key, nsems, semflg)) == -1) {
perror("semget: semget failed");
exit(1); }
else
   ...
```

25.2 Controlling Semaphores

`semctl()` changes permissions and other characteristics of a semaphore set. It is prototyped as follows:

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

It must be called with a valid semaphore ID, `semid`. The `semnum` value selects a semaphore within an array by its index. The `cmd` argument is one of the following control flags:

`GETVAL` — Return the value of a single semaphore.

`SETVAL` — Set the value of a single semaphore. In this case, arg is taken as arg.val, an int.

`GETPID` — Return the `PID` of the process that performed the last operation on the semaphore or array.

`GETNCNT` — Return the number of processes waiting for the value of a semaphore to increase.

`GETZCNT` — Return the number of processes waiting for the value of a particular semaphore to reach zero.

`GETALL` — Return the values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts (see below).

`SETALL` — Set values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts.

`IPC_STAT` — Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by `arg.buf`, a pointer to a buffer of type `semid_ds`.

`IPC_SET` — Set the effective user and group identification and permissions. In this case, `arg` is taken as `arg.buf`.

`IPC_RMID` — Remove the specified semaphore set.

A process must have an effective user identification of owner, creator, or superuser to perform an `IPC_SET` or `IPC_RMID` command. Read and write permission is required as for the other control commands. The following code illustrates `semctl ()`.

The fourth argument `union semun arg` is optional, depending upon the operation requested. If required it is of type `union semun`, which must be *explicitly* declared by the application program as:

```
          union semun {
               int val;
               struct semid_ds *buf;
               ushort *array;
          } arg;
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
               int val;
               struct semid_ds *buf;
               ushort *array;
          } arg;

int i;
int semnum = ....;
int cmd = GETALL; /* get value */




...
i = semctl(semid, semnum, cmd, arg);
if (i == -1) {
perror("semctl: semctl failed");
```

```
  exit(1);
 }
else
...
```

25.3 Semaphore Operations

`semop()` performs operations on a semaphore set. It is prototyped by:

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

The `semid` argument is the semaphore ID returned by a previous `semget()` call. The `sops` argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

- The semaphore number
- The operation to be performed
- Control flags, if any.

The `sembuf` structure specifies a semaphore operation, as defined in <`sys/sem.h`>.

```
struct sembuf {
        ushort_t        sem_num;        /* semaphore number */
        short           sem_op;         /* semaphore operation */
        short           sem_flg;        /* operation flags */
};
```

The `nsops` argument specifies the length of the array, the maximum size of which is determined by the `SEMOPM` configuration option; this is the maximum number of operations allowed by a single semop() call, and is set to 10 by default. The operation to be performed is determined as follows:

- A positive integer increments the semaphore value by that amount.
- A negative integer decrements the semaphore value by that amount. An attempt to set a semaphore to a value less than zero fails or blocks, depending on whether `IPC_NOWAIT` is in effect.

- A value of zero means to wait for the semaphore value to reach zero.

There are two control flags that can be used with `semop()`:

`IPC_NOWAIT` — Can be set for any operations in the array. Makes the function return without changing any semaphore value if any operation for which `IPC_NOWAIT` is set cannot be performed. The function fails if it tries to decrement a semaphore more than its current value, or tests a nonzero semaphore to be equal to zero.

`SEM_UNDO` — Allows individual operations in the array to be undone when the process exits.

This function takes a pointer, `sops`, to an array of semaphore operation structures. Each structure in the array contains data about an operation to perform on a semaphore. Any process with read permission can test whether a semaphore has a zero value. To increment or decrement a semaphore requires write permission. When an operation fails, none of the semaphores is altered.

The process blocks (unless the `IPC_NOWAIT` flag is set), and remains blocked until:

- the semaphore operations can all finish, so the call succeeds,
- the process receives a signal, or
- the semaphore set is removed.

Only one process at a time can update a semaphore. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a `semop()` call, no updates are done until all operations on the array can finish successfully.

If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this, the `SEM_UNDO` control flag makes `semop()` allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state. If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with `SEM_UNDO` in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state. When performing a semaphore operation with `SEM_UNDO` in effect, you must also have it in effect for the call that will perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This ensures that, unless the process is aborted, the values applied to the undo structure are cancel to zero. When the undo structure reaches zero, it is removed.

NOTE: Using `SEM_UNDO` inconsistently can lead to excessive resource consumption because allocated undo structures might not be freed until the system is rebooted.

The following code illustrates the `semop()` function:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

...
int i;
int nsops; /* number of operations to do */
int semid; /* semid of semaphore set */
struct sembuf *sops; /* ptr to operations to perform */

...

if ((semid = semop(semid, sops, nsops)) == -1)
{
perror("semop: semop failed");
 exit(1);
}
else
(void) fprintf(stderr, "semop: returned %d\n", i);
...
```

25.4 POSIX Semaphores: <semaphore.h>

POSIX semaphores are much lighter weight than are System V semaphores. A POSIX semaphore structure defines a single semaphore, not an array of up to twenty five semaphores. The POSIX semaphore functions are:

`sem_open()` — Connects to, and optionally creates, a named semaphore

`sem_init()` — Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_close()` — Ends the connection to an open semaphore.

`sem_unlink()` — Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

`sem_destroy()` — Initializes a semaphore structure (internal to the calling program, so not a named semaphore).

`sem_getvalue()` — Copies the value of the semaphore into the specified integer.

`sem_wait(), sem_trywait()` — Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process.

`sem_post()` — Increments the count of the semaphore.

The basic operation of these functions is essence the same as described above, except note there are more specialised functions, here. These are not discussed further here and the reader is referred to the online `man` pages for further details.

25.5 `semaphore.c`: Illustration of simple semaphore passing

```
/* semaphore.c --- simple illustration of dijkstra's semaphore analogy
 *
 *   We fork() a  child process so that we have two processes running:
 *   Each process communicates via a semaphore.
 *   The respective process can only do its work (not much here)
 *   When it notices that the semaphore track is free when it returns to 0
 *   Each process must modify the semaphore accordingly
 */

 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/sem.h>

 union semun {
               int val;
               struct semid_ds *buf;
               ushort *array;
          };

main()
{ int i,j;
  int pid;
  int semid; /* semid of semaphore set */
  key_t key = 1234; /* key to pass to semget() */
  int semflg = IPC_CREAT | 0666; /* semflg to pass to semget() */
  int nsems = 1; /* nsems to pass to semget() */
  int nsops; /* number of operations to do */
  struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));
  /* ptr to operations to perform */

  /* set up semaphore */

  (void) fprintf(stderr, "\nsemget: Setting up seamaphore: semget(%#lx, %\
%#o)\n",key, nsems, semflg);
   if ((semid = semget(key, nsems, semflg)) == -1) {
perror("semget: semget failed");
exit(1);
      } else
(void) fprintf(stderr, "semget: semget succeeded: semid =\
%d\n", semid);


  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
```

```
    }

if (pid == 0)
     { /* child */
       i = 0;


       while (i  < 3) {/* allow for 3 semaphore sets */

       nsops = 2;

       /* wait for semaphore to reach zero */

       sops[0].sem_num = 0; /* We only use one track */
       sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
       sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


       sops[1].sem_num = 0;
       sops[1].sem_op = 1; /* increment semaphore -- take control of track
       sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

       /* Recap the call to be made. */

       (void) fprintf(stderr,"\nsemop:Child  Calling semop(%d, &sops, %d) w
       for (j = 0; j < nsops; j++)
{
  (void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num)
  (void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
  (void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
}

       /* Make the semop() call and report the results. */
if ((j = semop(semid, sops, nsops)) == -1) {
perror("semop: semop failed");
}
   else
             {
```

```
(void) fprintf(stderr, "\tsemop: semop returned %d\n", j);

(void) fprintf(stderr, "\n\nChild Process Taking Control of Track: %d/3 times\n", i
sleep(5); /* DO Nothing for 5 seconds */

        nsops = 1;

        /* wait for semaphore to reach zero */
        sops[0].sem_num = 0;
                sops[0].sem_op = -1; /* Give UP COntrol of track */
        sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, asynchronou


               if ((j = semop(semid, sops, nsops)) == -1) {
perror("semop: semop failed");
}
   else
      (void) fprintf(stderr, "Child Process Giving up Control of Track: %d/3 times\
               sleep(5); /* halt process to allow parent to catch semaphor change f
              }
        ++i;
      }

     }
  else /* parent */
     {  /* pid hold id of child */

        i = 0;


       while (i  < 3) { /* allow for 3 semaphore sets */

       nsops = 2;

       /* wait for semaphore to reach zero */
       sops[0].sem_num = 0;
       sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
       sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */
       sops[1].sem_num = 0;
       sops[1].sem_op = 1; /* increment semaphore -- take control of track
       sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

       /* Recap the call to be made. */

       (void) fprintf(stderr,"\nsemop:Parent Calling semop(%d, &sops, %d) w
       for (j = 0; j < nsops; j++)
{
  (void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num)
  (void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
  (void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
}

       /* Make the semop() call and report the results. */
if ((j = semop(semid, sops, nsops)) == -1) {
perror("semop: semop failed");
}
   else
             {
(void) fprintf(stderr, "semop: semop returned %d\n", j);

(void) fprintf(stderr, "Parent Process Taking Control of Track: %d/3 times\
sleep(5); /* Do nothing for 5 seconds */

        nsops = 1;

        /* wait for semaphore to reach zero */
        sops[0].sem_num = 0;
                sops[0].sem_op = -1; /* Give UP COntrol of track */
        sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, asy

        if ((j = semop(semid, sops, nsops)) == -1) {
perror("semop: semop failed");
}
   else
```

```
      (void) fprintf(stderr, "Parent Process Giving up Control of Track: %d/3 time
              sleep(5); /* halt process to allow child to catch semaphor change f
             }
        ++i;

      }

     }
}
```

The key elements of this program are as follows:

- After a semaphore is created with as simple key `1234`, two prcesses are forked.
- Each process (parent and child) essentially performs the same operations:
  - Each process accesses the same semaphore *track* (`sops[].sem_num = 0`).
  - Each process waits for the *track* to become free and then attempts to take control of *track*

    This is achieved by setting appropriate `sops[].sem_op` values in the array.
  - Once the process has control it sleeps for 5 seconds (in reality some processing would take place in place of this simple illustration)
  - The process then gives up control of the *track* `sops[1].sem_op = -1`
  - an additional sleep operation is then performed to ensure that the other process has time to access the semaphore before a subsequent (same process) semaphore read.

    Note: There is no synchronisation here in this simple example an we have no control over how the OS will schedule the processes.

25.6 Some further example semaphore programs

The following suite of programs can be used to investigate interactively a variety of semaphore ideas (see exercises below).

The semaphore **must** be initialised with the `semget.c` program. The effects of controlling the semaphore queue and sending and receiving semaphore can be investigated with `semctl.c` and `semop.c` respectively.

25.6.1 `semget.c`: Illustrate the `semget()` function

```
/*
 * semget.c: Illustrate the semget() function.
 *
 * This is a simple exerciser of the semget() function. It prompts
 * for the arguments, makes the call, and reports the results.
*/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>

extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* key to pass to semget() */
 int  semflg;   /* semflg to pass to semget() */
 int  nsems;   /* nsems to pass to semget() */
 int  semid;   /* return value from semget() */

 (void) fprintf(stderr,
  "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 (void) fprintf(stderr, "Enter nsems value: ");
 (void) scanf("%i", &nsems);
 (void) fprintf(stderr, "\nExpected flags for semflg are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter semflg value: ");
 (void) scanf("%i", &semflg);
 (void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %
     %#o)\n",key, nsems, semflg);
 if ((semid = semget(key, nsems, semflg)) == -1) {
  perror("semget: semget failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "semget: semget succeeded: semid =
%d\n",
   semid);
  exit(0);
 }
}
```

25.6.2 semctl.c: Illustrate the semctl() function

```
/*
 * semctl.c:   Illustrate the semctl() function.
 *
 * This is a simple exerciser of the semctl() function. It lets you
 * perform one control operation on one semaphore set. It gives up
 * immediately if any control operation fails, so be careful not
to
 * set permissions to preclude read permission; you won't be able
to
 * reset the permissions with this code if you do.
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include    <time.h>

struct semid_ds semid_ds;

static void   do_semctl();
static void   do_stat();
extern char   *malloc();
extern void   exit();
extern void   perror();

char    warning_message[] = "If you remove read permission\
    for yourself, this program will fail frequently!";

main()
{
 union semun    arg;    /* union to pass to semctl() */
 int    cmd,    /* command to give to semctl() */
     i,    /* work area */
     semid,    /* semid to pass to semctl() */
     semnum;    /* semnum to pass to semctl() */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "Enter semid value: ");
 (void) scanf("%i", &semid);

 (void) fprintf(stderr, "Valid semctl cmd values are:\n");
 (void) fprintf(stderr, "\tGETALL = %d\n", GETALL);
 (void) fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
 (void) fprintf(stderr, "\tGETPID = %d\n", GETPID);
 (void) fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
 (void) fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\tSETALL = %d\n", SETALL);
 (void) fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
 (void) fprintf(stderr, "\nEnter cmd: ");
 (void) scanf("%i", &cmd);

 /* Do some setup operations needed by multiple commands. */
 switch (cmd) {
  case GETVAL:
  case SETVAL:
  case GETNCNT:
  case GETZCNT:
   /* Get the semaphore number for these commands. */
   (void) fprintf(stderr, "\nEnter semnum value: ");
   (void) scanf("%i", &semnum);
   break;
  case GETALL:
  case SETALL:
   /* Allocate a buffer for the semaphore values. */
   (void) fprintf(stderr,
    "Get number of semaphores in the set.\n");
   arg.buf = &semid_ds;
   do_semctl(semid, 0, IPC_STAT, arg);
   if (arg.array =
    (ushort *)malloc((unsigned)
     (semid_ds.sem_nsems * sizeof(ushort)))) {
    /* Break out if you got what you needed. */
    break;
   }
   (void) fprintf(stderr,
    "semctl: unable to allocate space for %d values\n",
    semid_ds.sem_nsems);
   exit(2);
 }

 /* Get the rest of the arguments needed for the specified
    command. */
 switch (cmd) {
  case SETVAL:
   /* Set value of one semaphore. */
   (void) fprintf(stderr, "\nEnter semaphore value: ");
   (void) scanf("%i", &arg.val);
   do_semctl(semid, semnum, SETVAL, arg);
   /* Fall through to verify the result. */
   (void) fprintf(stderr,
    "Do semctl GETVAL command to verify results.\n");
  case GETVAL:
   /* Get value of one semaphore. */
   arg.val = 0;
   do_semctl(semid, semnum, GETVAL, arg);
   break;
  case GETPID:
   /* Get PID of last process to successfully complete a
      semctl(SETVAL), semctl(SETALL), or semop() on the
      semaphore. */
   arg.val = 0;
   do_semctl(semid, 0, GETPID, arg);
   break;
  case GETNCNT:
   /* Get number of processes waiting for semaphore value to
      increase. */
   arg.val = 0;
```

```
   do_semctl(semid, semnum, GETNCNT, arg);
   break;
  case GETZCNT:
   /* Get number of processes waiting for semaphore value to
      become zero. */
   arg.val = 0;
   do_semctl(semid, semnum, GETZCNT, arg);
   break;
  case SETALL:
   /* Set the values of all semaphores in the set. */
   (void) fprintf(stderr,
       "There are %d semaphores in the set.\n",
       semid_ds.sem_nsems);
   (void) fprintf(stderr, "Enter semaphore values:\n");
   for (i = 0; i < semid_ds.sem_nsems; i++) {
    (void) fprintf(stderr, "Semaphore %d: ", i);
    (void) scanf("%hi", &arg.array[i]);
   }
   do_semctl(semid, 0, SETALL, arg);
   /* Fall through to verify the results. */
   (void) fprintf(stderr,
    "Do semctl GETALL command to verify results.\n");
  case GETALL:
   /* Get and print the values of all semaphores in the
      set.*/
   do_semctl(semid, 0, GETALL, arg);
   (void) fprintf(stderr,
       "The values of the %d semaphores are:\n",
       semid_ds.sem_nsems);
   for (i = 0; i < semid_ds.sem_nsems; i++)
    (void) fprintf(stderr, "%d ", arg.array[i]);
   (void) fprintf(stderr, "\n");
   break;
  case IPC_SET:
   /* Modify mode and/or ownership. */
   arg.buf = &semid_ds;
   do_semctl(semid, 0, IPC_STAT, arg);
   (void) fprintf(stderr, "Status before IPC_SET:\n");
```

```
    do_stat();
    (void) fprintf(stderr, "Enter sem_perm.uid value: ");
    (void) scanf("%hi", &semid_ds.sem_perm.uid);
    (void) fprintf(stderr, "Enter sem_perm.gid value: ");
    (void) scanf("%hi", &semid_ds.sem_perm.gid);
    (void) fprintf(stderr, "%s\n", warning_message);
    (void) fprintf(stderr, "Enter sem_perm.mode value: ");
    (void) scanf("%hi", &semid_ds.sem_perm.mode);
    do_semctl(semid, 0, IPC_SET, arg);
    /* Fall through to verify changes. */
    (void) fprintf(stderr, "Status after IPC_SET:\n");
   case IPC_STAT:
    /* Get and print current status. */
    arg.buf = &semid_ds;
    do_semctl(semid, 0, IPC_STAT, arg);
    do_stat();
    break;
   case IPC_RMID:
    /* Remove the semaphore set. */
    arg.val = 0;
    do_semctl(semid, 0, IPC_RMID, arg);
    break;
   default:
    /* Pass unknown command to semctl. */
    arg.val = 0;
    do_semctl(semid, 0, cmd, arg);
    break;
 }
 exit(0);
}

/*
 * Print indication of arguments being passed to semctl(), call
 * semctl(), and report the results. If semctl() fails, do not
 * return; this example doesn't deal with errors, it just reports
 * them.
 */
static void
```

```
do_semctl(semid, semnum, cmd, arg)
union semun  arg;
int  cmd,
  semid,
  semnum;
{
 register int      i;   /* work area */

 void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d,
",
    semid, semnum, cmd);
 switch (cmd) {
  case GETALL:
   (void) fprintf(stderr, "arg.array = %#x)\n",
       arg.array);
   break;
  case IPC_STAT:
  case IPC_SET:
   (void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
   break;
  case SETALL:
   (void) fprintf(stderr, "arg.array = [", arg.buf);
   for (i = 0;i < semid_ds.sem_nsems;) {
    (void) fprintf(stderr, "%d", arg.array[i++]);
    if (i < semid_ds.sem_nsems)
      (void) fprintf(stderr, ", ");
   }
   (void) fprintf(stderr, "])\n");
   break;
  case SETVAL:
  default:
   (void) fprintf(stderr, "arg.val = %d)\n", arg.val);
   break;
 }
 i = semctl(semid, semnum, cmd, arg);
 if (i == -1) {
  perror("semctl: semctl failed");
  exit(1);
 }
 (void) fprintf(stderr, "semctl: semctl returned %d\n", i);
 return;
}

/*
 * Display contents of commonly used pieces of the status
structure.
 */
static void
do_stat()
{
 (void) fprintf(stderr, "sem_perm.uid = %d\n",
       semid_ds.sem_perm.uid);
 (void) fprintf(stderr, "sem_perm.gid = %d\n",
       semid_ds.sem_perm.gid);
 (void) fprintf(stderr, "sem_perm.cuid = %d\n",
       semid_ds.sem_perm.cuid);
 (void) fprintf(stderr, "sem_perm.cgid = %d\n",
       semid_ds.sem_perm.cgid);
 (void) fprintf(stderr, "sem_perm.mode = %#o, ",
       semid_ds.sem_perm.mode);
 (void) fprintf(stderr, "access permissions = %#o\n",
       semid_ds.sem_perm.mode & 0777);
 (void) fprintf(stderr, "sem_nsems = %d\n",
semid_ds.sem_nsems);
 (void) fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
      ctime(&semid_ds.sem_otime) : "Not Set\n");
 (void) fprintf(stderr, "sem_ctime = %s",
       ctime(&semid_ds.sem_ctime));
}
```

25.6.3 `semop()` Sample Program to Illustrate `semop()`

```
/*
 * semop.c: Illustrate the semop() function.
 *
 * This is a simple exerciser of the semop() function. It lets you
 * to set up arguments for semop() and make the call. It then
reports
 * the results repeatedly on one semaphore set. You must have read
 * permission on the semaphore set or this exerciser will fail.
(It
 * needs read permission to get the number of semaphores in the set
 * and to report the values before and after calls to semop().)
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

static int      ask();
extern void     exit();
extern void     free();
extern char     *malloc();
extern void     perror();

static struct semid_ds  semid_ds;           /* status of semaphore set */

static char    error_mesg1[] = "semop: Can't allocate space for %d\
        semaphore values. Giving up.\n";
static char   error_mesg2[] = "semop: Can't allocate space for %d\
        sembuf structures. Giving up.\n";

main()
{
 register int    i;   /* work area */
 int    nsops;   /* number of operations to do */
 int    semid;   /* semid of semaphore set */
 struct sembuf    *sops;   /* ptr to operations to perform */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
```

```
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
/* Loop until the invoker doesn't want to do anymore. */
while (nsops = ask(&semid, &sops)) {
 /* Initialize the array of operations to be performed.*/
 for (i = 0; i < nsops; i++) {
  (void) fprintf(stderr,
    "\nEnter values for operation %d of %d.\n",
      i + 1, nsops);
  (void) fprintf(stderr,
    "sem_num(valid values are 0 <= sem_num < %d): ",
    semid_ds.sem_nsems);
  (void) scanf("%hi", &sops[i].sem_num);
  (void) fprintf(stderr, "sem_op: ");
  (void) scanf("%hi", &sops[i].sem_op);
  (void) fprintf(stderr,
    "Expected flags in sem_flg are:\n");
  (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",
    IPC_NOWAIT);
  (void) fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",
    SEM_UNDO);
  (void) fprintf(stderr, "sem_flg: ");
  (void) scanf("%hi", &sops[i].sem_flg);
 }

 /* Recap the call to be made. */
 (void) fprintf(stderr,
    "\nsemop: Calling semop(%d, &sops, %d) with:",
    semid, nsops);
 for (i = 0; i < nsops; i++)
 {
  (void) fprintf(stderr, "\nsops[%d].sem_num = %d, ", i,
     sops[i].sem_num);
  (void) fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
  (void) fprintf(stderr, "sem_flg = %#o\n",
     sops[i].sem_flg);
 }
```

```
  /* Make the semop() call and report the results. */
  if ((i = semop(semid, sops, nsops)) == -1) {
   perror("semop: semop failed");
  } else {
   (void) fprintf(stderr, "semop: semop returned %d\n", i);
  }
 }
}

/*
 * Ask if user wants to continue.
 *
 * On the first call:
 * Get the semid to be processed and supply it to the caller.
 * On each call:
 *  1. Print current semaphore values.
 *  2. Ask user how many operations are to be performed on the next
 *     call to semop. Allocate an array of sembuf structures
 *     sufficient for the job and set caller-supplied pointer to
that
 *     array. (The array is reused on subsequent calls if it is big
 *     enough. If it isn't, it is freed and a larger array is
 *     allocated.)
 */
static
ask(semidp, sopsp)
int   *semidp;  /* pointer to semid (used only the first time) */
struct sembuf   **sopsp;
{
 static union semun     arg;  /* argument to semctl */
 int     i;  /* work area */
 static int     nsops = 0;  /* size of currently allocated
           sembuf array */
 static int     semid = -1;   /* semid supplied by user */
 static struct sembuf    *sops;     /* pointer to allocated array */

 if (semid < 0) {
  /* First call; get semid from user and the current state of
```

```
    the semaphore set. */
  (void) fprintf(stderr,
    "Enter semid of the semaphore set you want to use: ");
  (void) scanf("%i", &semid);
  *semidp = semid;
  arg.buf = &semid_ds;
  if (semctl(semid, 0, IPC_STAT, arg) == -1) {
   perror("semop: semctl(IPC_STAT) failed");
   /* Note that if semctl fails, semid_ds remains filled
      with zeros, so later test for number of semaphores will
      be zero. */
   (void) fprintf(stderr,
     "Before and after values are not printed.\n");
  } else {
   if ((arg.array = (ushort *)malloc(
    (unsigned)(sizeof(ushort) * semid_ds.sem_nsems)))
       == NULL) {
    (void) fprintf(stderr, error_mesg1,
       semid_ds.sem_nsems);
    exit(1);
   }
  }
 }
 /* Print current semaphore values. */
 if (semid_ds.sem_nsems) {
  (void) fprintf(stderr,
      "There are %d semaphores in the set.\n",
      semid_ds.sem_nsems);
  if (semctl(semid, 0, GETALL, arg) == -1) {
   perror("semop: semctl(GETALL) failed");
  } else {
   (void) fprintf(stderr, "Current semaphore values are:");
   for (i = 0; i < semid_ds.sem_nsems;
    (void) fprintf(stderr, " %d", arg.array[i++]));
   (void) fprintf(stderr, "\n");
  }
 }
 /* Find out how many operations are going to be done in the
```

```
next
    call and allocate enough space to do it. */
 (void) fprintf(stderr,
     "How many semaphore operations do you want %s\n",
     "on the next call to semop()?");
 (void) fprintf(stderr, "Enter 0 or control-D to quit: ");
 i = 0;
 if (scanf("%i", &i) == EOF || i == 0)
  exit(0);
 if (i > nsops) {
  if (nsops)
   free((char *)sops);
  nsops = i;
  if ((sops = (struct sembuf *)malloc((unsigned)(nsops *
   sizeof(struct sembuf)))) == NULL) {
   (void) fprintf(stderr, error_mesg2, nsops);
   exit(2);
  }
 }
 *sopsp = sops;
 return (i);
}
```

25.7 Exercises

Exercise 25.1 *Write 2 programs that will communicate* **both ways** *(*i.e *each process can read and write) when run concurrently via semaphores.*

Exercise 25.2 *Modify the* `semaphore.c` *program to handle synchronous semaphore communication semaphores.*

Exercise 25.3 *Write 3 programs that communicate together via semaphores according to the following specifications:*

`sem_server.c` — *a program that can communicate independently (on different semaphore tracks) with two clients programs.*

`sem_client1.c` — *a program that talks to* `sem_server.c` *on one track.*

`sem_client2.c` — *a program that talks to* `sem_server.c` *on another track to* `sem_client1.c`*.*

Exercise 25.4 *Compile the programs* `semget.c`*,* `semctl.c` *and* `semop.c` *and then*

- *investigate and understand fully the operations of the flags (access, creation* etc. *permissions) you can set interactively in the programs.*
- *Use the prgrams to:*
  - *Send and receive semaphores of 3 different semaphore* `track`*s.*
  - *Inquire about the state of the semaphore queue with* `semctl.c`*. Add/delete a few semaphores (using* `semop.c` *and perform the inquiry once more.*
  - *Use* `semctl.c` *to alter a semaphore on the queue.*
  - *Use* `semctl.c` *to delete a semaphore from the queue.*

Chapter 26

IPC:Shared Memory

Shared Memory is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

In the Solaris 2.x operating system, the most efficient way to implement shared memory applications is to rely on the `mmap()` function and on the system's native virtual memory facility. Solaris 2.x also supports System V shared memory, which is another way to let multiple processes attach a segment of physical memory to their virtual address spaces. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using `shmget()|`. The original owner of a shared memory segment can assign ownership to another user with `shmctl()`. It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`. Once created, a shared segment can be attached to a process address space using `shmat()`. It can be detached using `shmdt()` (see `shmop()`). The attaching process must have the appropriate permissions for `shmat()`. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypews can be found in <`sys/shm.h`>.

26.1 Accessing a Shared Memory Segment

`shmget()` is used to obtain access to a shared memory segment. It is prottyped by:

```
int shmget(key_t key, size_t size, int shmflg);
```

The `key` argument is a access value associated with the semaphore ID. The `size` argument is the size in bytes of the requested shared memory. The `shmflg` argument specifies the initial access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID. This call is also used to get the ID of an existing shared segment (from a process requesting sharing of some existing memory portion).

The following code illustrates `shmget()`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

...

key_t key; /* key to be passed to shmget() */
int shmflg; /* shmflg to be passed to shmget() */
int shmid; /* return value from shmget() */
int size; /* size to be passed to shmget() */

...

key = ...
size = ...
shmflg) = ...

if ((shmid = shmget (key, size, shmflg)) == -1) {
   perror("shmget: shmget failed"); exit(1); } else {
   (void) fprintf(stderr, "shmget: shmget returned %d\n", shmid);
   exit(0);
  }
...
```

26.1.1 Controlling a Shared Memory Segment

`shmctl()` is used to alter the permissions and other characteristics of a shared memory segment. It is prototyped as follows:

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The process must have an effective `shmid` of owner, creator or superuser to perform this command. The `cmd` argument is one of following control commands:

`SHM_LOCK` — Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.

`SHM_UNLOCK` — Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.

`IPC_STAT` — Return the status information contained in the control structure and place it in the buffer pointed to by buf. The process must have read permission on the segment to perform this command.

`IPC_SET` — Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.

`IPC_RMID` — Remove the shared memory segment.

The `buf` is a sructure of type `struct shmid_ds` which is defined in <`sys/shm.h`>
The following code illustrates `shmctl()`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

...

int cmd; /* command code for shmctl() */
int shmid; /* segment ID */
struct shmid_ds shmid_ds; /* shared memory data structure to
```

```
                              hold results */
...

shmid = ...
cmd = ...
if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
    perror("shmctl: shmctl failed");
    exit(1);
   }
...
```

26.2 Attaching and Detaching a Shared Memory Segment

shmat() and shmdt() are used to attach and detach shared memory segments. They are prototypes as follows:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);
```

shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr

. The following code illustrates calls to shmat() and shmdt():

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

static struct state { /* Internal record of attached segments. */
          int shmid; /* shmid of attached segment */
          char *shmaddr; /* attach point */
          int shmflg; /* flags used on attach */
         } ap[MAXnap]; /* State of current attached segments. */
int nap; /* Number of currently attached segments. */

...
```

```
char *addr; /* address work variable */
register int i; /* work area */
register struct state *p; /* ptr to current state entry */
...

p = &ap[nap++];
p->shmid = ...
p->shmaddr = ...
p->shmflg = ...

p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
if(p->shmaddr == (char *)-1) {
     perror("shmop: shmat failed");
     nap--;
    } else
    (void) fprintf(stderr, "shmop: shmat returned %#8.8x\n",
p->shmaddr);

...
i = shmdt(addr);
if(i == -1) {
    perror("shmop: shmdt failed");
    } else {
  (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);

for (p = ap, i = nap; i--; p++)
  if (p->shmaddr == addr) *p = ap[--nap];

}
...
```

26.3 Example two processes comunicating via shared memory:`shm_server.c`, `shm_client.c`

We develop two programs here that illustrate the passing of a simple piece of memery (a string) between the processes if running simulatenously:

`shm_server.c` — simply creates the string and shared memory portion.

`shm_client.c` — attaches itself to the created shared memory portion and uses the string (`printf`.

The code listings of the 2 programs no follow:

26.3.1 `shm_server.c`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment
     * "5678".
     */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now put some things into the memory for the
     * other process to read.
     */
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    /*
     * Finally, we wait until the other process
     * changes the first character of our memory
     * to '*', indicating that it has read what
     * we put there.
     */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

26.3.2 shm_client.c

```
/*
 * shm-client - client program to demonstrate shared memory.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now read what the server put in the memory.
     */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
}
```

26.4 POSIX Shared Memory

POSIX shared memory is actually a variation of mapped memory. The major differences are to use `shm_open()` to open the shared memory object (instead of calling `open()`) and use `shm_unlink()` to close and delete the object (instead of calling `close()` which does not remove the object). The options in `shm_open()` are substantially fewer than the number of options provided in `open()`.

26.5 Mapped memory

In a system with fixed memory (non-virtual), the address space of a process occupies and is limited to a portion of the system's main memory. In Solaris 2.x virtual memory the actual address space of a process occupies a file in the swap partition of disk storage (the file is called the backing store). Pages of main memory buffer the active (or recently active) portions of the process address space to provide code for the CPU(s) to execute and data for the program to process.

A page of address space is loaded when an address that is not currently in memory is accessed by a CPU, causing a page fault. Since execution cannot continue until the page fault is resolved by reading the referenced address segment into memory, the process sleeps until the page has been read. The most obvious difference between the two memory systems for the application developer is that virtual memory lets applications occupy much larger address spaces. Less obvious advantages of virtual memory are much simpler and more efficient file I/O and very efficient sharing of memory between processes.

26.5.1 Address Spaces and Mapping

Since backing store files (the process address space) exist only in swap storage, they are not included in the UNIX named file space. (This makes backing store files inaccessible to other processes.) However, it is a simple extension to allow the logical insertion of all, or part, of one, or more, named files in the backing store and to treat the result as a single address space. This is called mapping. With mapping, any part of any readable or writable file can be logically included in a process's address space. Like any other portion of the process's address space, no page of the file is not actually loaded into memory until a page fault forces this action. Pages of memory are written to the file only if their contents have been modified. So, reading from and writing to files is completely automatic and very efficient. More than one process can map a single named file. This provides very efficient memory sharing between processes. All or part of other files can also be shared between processes.

Not all named file system objects can be mapped. Devices that cannot be treated as storage, such as terminal and network device files, are examples of objects that cannot be mapped. A process address space is defined by all of the files (or portions of files) mapped into the address space. Each mapping is sized and aligned to the page boundaries of the system on which the process is executing. There is no memory associated with processes themselves.

A process page maps to only one object at a time, although an object address may be the subject of many process mappings. The notion of a ”page” is not a property of the mapped object. Mapping an object only provides the potential for a process to read or write the object's contents. Mapping makes the object's contents directly addressable by a process. Applications can access the storage resources they use directly rather than indirectly through read and write. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the read, modify buffer, write cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

Because the file system name space includes any directory trees that are connected from other systems via NFS, any networked file can also be mapped into a process's address space.

26.5.2 Coherence

Whether to share memory or to share data contained in the file, when multiple process map a file simultaneously there may be problems with simultaneous access to data elements. Such processes can cooperate through any of the synchronization mechanisms provided in Solaris 2.x. Because they are very light weight, the most efficient synchronization mechanisms in Solaris 2.x are the threads library ones.

26.5.3 Creating and Using Mappings

`mmap()` establishes a mapping of a named file system object (or part of one) into a process address space. It is the basic memory management function and it is very simple.

- First `open()` the file, then
- `mmap()` it with appropriate access and sharing options
- Away you go.

`mmap` is prototypes as follows:

```
#include <sys/types.h>
#include <sys/mman.h>

caddr_t mmap(caddr_t addr, size_t len, int prot, int flags,
          int fildes, off_t off);
```

The mapping established by `mmap()` replaces any previous mappings for specified address range. The `flags` `MAP_SHARED` and `MAP_PRIVATE` specify the mapping type, and one of them must be specified. `MAP_SHARED` specifies that writes modify the mapped object. No further operations on the object are needed to make the change. `MAP_PRIVATE` specifies that an initial write to the mapped area creates a copy of the page and all writes reference the copy. Only modified pages are copied.

A mapping type is retained across a `fork()`. The file descriptor used in a mmap call need not be kept open after the mapping is established. If it is closed, the mapping remains until the mapping is undone by `munmap()` or be replacing in with a new mapping. If a mapped file is shortened by a call to truncate, an access to the area of the file that no longer exists causes a `SIGBUS` signal.

The following code fragment demonstrates a use of this to create a block of scratch storage in a program, at an address that the system chooses.:

```
int fd;
caddr_t result;
if ((fd = open("/dev/zero", O_RDWR)) == -1)
   return ((caddr_t)-1);

result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
(void) close(fd);
```

26.5.4 Other Memory Control Functions

`int mlock(caddr_t addr, size_t len)` causes the pages in the specified address range to be locked in physical memory. References to locked pages (in this or other processes) do not result in page faults that require an I/O operation. This operation ties up physical resources and can disrupt normal system operation, so, use of `mlock()` is limited to the superuser. The system lets only a configuration dependent limit of pages be locked in memory. The call to mlock fails if this limit is exceeded.

`int munlock(caddr_t addr, size_t len)` releases the locks on physical pages. If multiple `mlock()` calls are made on an address range of a single mapping, a single munlock call is release the locks. However, if different mappings to the same pages are mlocked, the pages are not unlocked until the locks on all the mappings are released. Locks are also released when a mapping is removed, either through being replaced with an mmap operation or removed with munmap. A lock is transferred between pages on the "copy-on-write' event associated with a `MAP_PRIVATE` mapping, thus locks on an address range that includes `MAP_PRIVATE` mappings will be retained transparently along with the copy-on-write redirection (see mmap above for a discussion of this redirection)

`int mlockall(int flags)` and `int munlockall(void)` are similar to `mlock()` and `munlock()`, but they operate on entire address spaces. `mlockall()` sets locks on all pages in the address space and `munlockall()` removes all locks on all pages in the address space, whether established by mlock or mlockall.

`int msync(caddr_t addr, size_t len, int flags)` causes all modified pages in the specified address range to be flushed to the objects mapped by those addresses. It is similar to `fsync()` for files.

`long sysconf(int name)` returns the system dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page. Note that it is not unusual for page sizes to vary even among implementations of the same instruction set.

`int mprotect(caddr_t addr, size_t len, int prot)` assigns the specified protection to all pages in the specified address range. The protection cannot exceed the permissions allowed on the underlying object.

`int brk(void *endds)` and `void *sbrk(int incr)` are called to add storage to the data segment of a process. A process can manipulate this area by calling `brk()` and `sbrk()`. `brk()` sets the system idea of the lowest data segment location not used by the caller to addr (rounded up to the next multiple of the system page size). `sbrk()` adds incr bytes to the caller data space and returns a pointer to the start of the new data area.

26.6 Some further example shared memory programs

The following suite of programs can be used to investigate interactively a variety of shared ideas (see exercises below).

The semaphore **must** be initialised with the `shmget.c` program. The effects of controlling shared memory and accessing can be investigated with `shmctl.c` and `shmop.c` respectively.

26.6.1 `shmget.c`:Sample Program to Illustrate shmget()

```
/*
 * shmget.c: Illustrate the shmget() function.
 *
 * This is a simple exerciser of the shmget() function. It
prompts
 * for the arguments, makes the call, and reports the results.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* key to be passed to shmget() */
 int  shmflg;   /* shmflg to be passed to shmget() */
 int  shmid;   /* return value from shmget() */
 int  size;   /* size to be passed to shmget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the key. */
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 /* Get the size of the segment. */
 (void) fprintf(stderr, "Enter size: ");
 (void) scanf("%i", &size);

 /* Get the shmflg value. */
 (void) fprintf(stderr,
    "Expected flags for the shmflg argument are:\n");
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n",
IPC_CREAT);
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter shmflg: ");
 (void) scanf("%i", &shmflg);

 /* Make the call and report the results. */
 (void) fprintf(stderr,
     "shmget: Calling shmget(%#lx, %d, %#o)\n",
     key, size, shmflg);
 if ((shmid = shmget (key, size, shmflg)) == -1) {
  perror("shmget: shmget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
     "shmget: shmget returned %d\n", shmid);
  exit(0);
 }
}
```

26.6.2 `shmctl.c`: Sample Program to Illustrate `shmctl()`

```
/*
 * shmctl.c: Illustrate the shmctl() function.
 *
 * This is a simple exerciser of the shmctl() function. It lets you
 * to perform one control operation on one shared memory segment.
 * (Some operations are done for the user whether requested or
not.
 * It gives up immediately if any control operation fails. Be
careful
 * not to set permissions to preclude read permission; you won't
be
 *able to reset the permissions with this code if you do.)
*/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
#include   <time.h>
static void   do_shmctl();
extern void   exit();
extern void   perror();

main()
{
 int  cmd;  /* command code for shmctl() */
 int  shmid;  /* segment ID */
 struct shmid_ds  shmid_ds;     /* shared memory data structure to
         hold results */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
     "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
```

```
/* Get shmid and cmd. */
(void) fprintf(stderr,
    "Enter the shmid for the desired segment: ");
(void) scanf("%i", &shmid);
(void) fprintf(stderr, "Valid shmctl cmd values are:\n");
(void) fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
(void) fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
(void) fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
(void) fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
(void) fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
(void) fprintf(stderr, "Enter the desired cmd value: ");
(void) scanf("%i", &cmd);

switch (cmd) {
 case IPC_STAT:
  /* Get shared memory segment status. */
  break;
 case IPC_SET:
  /* Set owner UID and GID and permissions. */
  /* Get and print current values. */
  do_shmctl(shmid, IPC_STAT, &shmid_ds);
  /* Set UID, GID, and permissions to be loaded. */
  (void) fprintf(stderr, "\nEnter shm_perm.uid: ");
  (void) scanf("%hi", &shmid_ds.shm_perm.uid);
  (void) fprintf(stderr, "Enter shm_perm.gid: ");
  (void) scanf("%hi", &shmid_ds.shm_perm.gid);
  (void) fprintf(stderr,
   "Note: Keep read permission for yourself.\n");
  (void) fprintf(stderr, "Enter shm_perm.mode: ");
  (void) scanf("%hi", &shmid_ds.shm_perm.mode);
  break;
 case IPC_RMID:
  /* Remove the segment when the last attach point is
     detached. */
  break;
 case SHM_LOCK:
  /* Lock the shared memory segment. */
```

```
   break;
  case SHM_UNLOCK:
   /* Unlock the shared memory segment. */
   break;
  default:
   /* Unknown command will be passed to shmctl. */
   break;
 }
 do_shmctl(shmid, cmd, &shmid_ds);
 exit(0);
}

/*
 * Display the arguments being passed to shmctl(), call shmctl(),
 * and report the results. If shmctl() fails, do not return; this
 * example doesn't deal with errors, it just reports them.
 */
static void
do_shmctl(shmid, cmd, buf)
int   shmid,   /* attach point */
   cmd;   /* command code */
struct shmid_ds   *buf;   /* pointer to shared memory data structure */
{
 register int    rtrn;  /* hold area */

 (void) fprintf(stderr, "shmctl: Calling shmctl(%d, %d,
buf)\n",
  shmid, cmd);
 if (cmd == IPC_SET) {
  (void) fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
     buf->shm_perm.uid);
  (void) fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
     buf->shm_perm.gid);
  (void) fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
     buf->shm_perm.mode);
 }
 if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
  perror("shmctl: shmctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
      "shmctl: shmctl returned %d\n", rtrn);
 }
 if (cmd != IPC_STAT && cmd != IPC_SET)
  return;

 /* Print the current status. */
 (void) fprintf(stderr, "\nCurrent status:\n");
 (void) fprintf(stderr, "\tshm_perm.uid = %d\n",
      buf->shm_perm.uid);
 (void) fprintf(stderr, "\tshm_perm.gid = %d\n",
      buf->shm_perm.gid);
 (void) fprintf(stderr, "\tshm_perm.cuid = %d\n",
      buf->shm_perm.cuid);
 (void) fprintf(stderr, "\tshm_perm.cgid = %d\n",
      buf->shm_perm.cgid);
 (void) fprintf(stderr, "\tshm_perm.mode = %#o\n",
      buf->shm_perm.mode);
 (void) fprintf(stderr, "\tshm_perm.key = %#x\n",
      buf->shm_perm.key);
 (void) fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
 (void) fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
 (void) fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
 (void) fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
 (void) fprintf(stderr, "\tshm_atime = %s",
   buf->shm_atime ? ctime(&buf->shm_atime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_dtime = %s",
   buf->shm_dtime ? ctime(&buf->shm_dtime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_ctime = %s",
      ctime(&buf->shm_ctime));
}
```

26.6.3 shmop.c: Sample Program to Illustrate shmat() and shmdt()

```
/*
 * shmop.c: Illustrate the shmat() and shmdt() functions.
 *
 * This is a simple exerciser for the shmat() and shmdt() system
 * calls. It allows you to attach and detach segments and to
 * write strings into and read strings from attached segments.
 */

#include   <stdio.h>
#include   <setjmp.h>
#include   <signal.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>

#define   MAXnap  4 /* Maximum number of concurrent attaches. */

static   ask();
static void  catcher();
extern void  exit();
static   good_addr();
extern void  perror();
extern char  *shmat();

static struct state     { /* Internal record of currently attached
segments. */
 int  shmid;   /* shmid of attached segment */
 char  *shmaddr;   /* attach point */
 int  shmflg;   /* flags used on attach */
} ap[MAXnap];     /* State of current attached segments. */

static int    nap;  /* Number of currently attached segments. */
static jmp_buf   segvbuf;   /* Process state save area for SIGSEGV
         catching. */
```

```
main()
{
 register int    action;   /* action to be performed */
 char    *addr;   /* address work area */
 register int    i;   /* work area */
 register struct state   *p;    /* ptr to current state entry */
 void   (*savefunc)();  /* SIGSEGV state hold area */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 while (action = ask()) {
  if (nap) {
   (void) fprintf(stderr,
      "\nCurrently attached segment(s):\n");
   (void) fprintf(stderr, " shmid address\n");
   (void) fprintf(stderr, "------ ----------\n");
   p = &ap[nap];
   while (p-- != ap) {
    (void) fprintf(stderr, "%6d", p->shmid);
    (void) fprintf(stderr, "%#11x", p->shmaddr);
    (void) fprintf(stderr, " Read%s\n",
     (p->shmflg & SHM_RDONLY) ?
     "-Only" : "/Write");
   }
  } else
   (void) fprintf(stderr,
    "\nNo segments are currently attached.\n");
  switch (action) {
  case 1:   /* Shmat requested. */
   /* Verify that there is space for another attach. */
   if (nap == MAXnap) {
    (void) fprintf(stderr, "%s %d %s\n",
       "This simple example will only allow",
       MAXnap, "attached segments.");
    break;
   }
   p = &ap[nap++];
   /* Get the arguments, make the call, report the
    results, and update the current state array. */
   (void) fprintf(stderr,
    "Enter shmid of segment to attach: ");
   (void) scanf("%i", &p->shmid);

   (void) fprintf(stderr, "Enter shmaddr: ");
   (void) scanf("%i", &p->shmaddr);
   (void) fprintf(stderr,
    "Meaningful shmflg values are:\n");
   (void) fprintf(stderr, "\tSHM_RDONLY = \t%#8.8o\n",
    SHM_RDONLY);
   (void) fprintf(stderr, "\tSHM_RND = \t%#8.8o\n",
    SHM_RND);
   (void) fprintf(stderr, "Enter shmflg value: ");
   (void) scanf("%i", &p->shmflg);

   (void) fprintf(stderr,
    "shmop: Calling shmat(%d, %#x, %#o)\n",
    p->shmid, p->shmaddr, p->shmflg);
   p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
   if(p->shmaddr == (char *)-1) {
    perror("shmop: shmat failed");
    nap--;
   } else {
    (void) fprintf(stderr,
     "shmop: shmat returned %#8.8x\n",
     p->shmaddr);
   }
   break;

  case 2:   /* Shmdt requested. */
   /* Get the address, make the call, report the results,
    and make the internal state match. */
   (void) fprintf(stderr,
    "Enter detach shmaddr: ");
   (void) scanf("%i", &addr);

   i = shmdt(addr);
   if(i == -1) {
    perror("shmop: shmdt failed");
   } else {
    (void) fprintf(stderr,
     "shmop: shmdt returned %d\n", i);
    for (p = ap, i = nap; i--; p++) {
     if (p->shmaddr == addr)
      *p = ap[--nap];
    }
   }
   break;
  case 3: /* Read from segment requested. */
   if (nap == 0)
    break;

   (void) fprintf(stderr, "Enter address of an %s",
    "attached segment: ");
   (void) scanf("%i", &addr);

   if (good_addr(addr))
    (void) fprintf(stderr, "String @ %#x is ‘%s’\n",
     addr, addr);
   break;

  case 4: /* Write to segment requested. */
   if (nap == 0)
    break;

   (void) fprintf(stderr, "Enter address of an %s",
    "attached segment: ");
   (void) scanf("%i", &addr);

   /* Set up SIGSEGV catch routine to trap attempts to
    write into a read-only attached segment. */
   savefunc = signal(SIGSEGV, catcher);
```

```
    if (setjmp(segvbuf)) {
     (void) fprintf(stderr, "shmop: %s: %s\n",
      "SIGSEGV signal caught",
      "Write aborted.");
    } else {
     if (good_addr(addr)) {
      (void) fflush(stdin);
      (void) fprintf(stderr, "%s %s %#x:\n",
       "Enter one line to be copied",
       "to shared segment attached @",
       addr);
      (void) gets(addr);
     }
    }
    (void) fflush(stdin);

    /* Restore SIGSEGV to previous condition. */
    (void) signal(SIGSEGV, savefunc);
    break;
  }
 }
 exit(0);
 /*NOTREACHED*/
}
/*
** Ask for next action.
*/
static
ask()
{
 int  response;   /* user response */
 do {
    (void) fprintf(stderr, "Your options are:\n");
    (void) fprintf(stderr, "\t^D = exit\n");
    (void) fprintf(stderr, "\t 0 = exit\n");
    (void) fprintf(stderr, "\t 1 = shmat\n");
    (void) fprintf(stderr, "\t 2 = shmdt\n");
```

```
   (void) fprintf(stderr, "\t 3 = read from segment\n");
   (void) fprintf(stderr, "\t 4 = write to segment\n");
   (void) fprintf(stderr,
    "Enter the number corresponding to your choice: ");

   /* Preset response so "^D" will be interpreted as exit. */
   response = 0;
   (void) scanf("%i", &response);
 } while (response < 0 || response > 4);
 return (response);
}
/*
** Catch signal caused by attempt to write into shared memory
segment
** attached with SHM_RDONLY flag set.
*/
/*ARGSUSED*/
static void
catcher(sig)
{
 longjmp(segvbuf, 1);
 /*NOTREACHED*/
}
/*
** Verify that given address is the address of an attached
segment.
** Return 1 if address is valid; 0 if not.
*/
static
good_addr(address)
char *address;
{
 register struct state          *p;   /* ptr to state of attached
segment */

 for (p = ap; p != &ap[nap]; p++)
   if (p->shmaddr == address)
    return(1);
 return(0);
}
```

26.7 Exercises

Exercise 26.1 *Write 2 programs that will communicate via shared memory and semaphores. Data will be exchanged via memory and semaphores will be used to synchronise and notify each process when operations such as memory loaded and memory read have been performed.*

Exercise 26.2 *Compile the programs* `shmget.c`*,* `shmctl.c` *and* `shmop.c` *and then*

- *investigate and understand fully the operations of the flags (access, creation* etc. *permissions) you can set interactively in the programs.*
- *Use the prgrams to:*
  - *Exchange data between two processe running as* `shmop.c`*.*
  - *Inquire about the state of shared memory with* `shmctl.c`*.*
  - *Use* `semctl.c` *to lock a shared memory segment.*
  - *Use* `semctl.c` *to delete a shared memory segment.*

Exercise 26.3 *Write 2 programs that will communicate via mapped memory.*

Chapter 27

IPC:Sockets

Sockets provide point-to-point, two-way communication between two processes. Sockets are very versatile and are a basic component of interprocess and intersystem communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.

Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. Twenty three socket domains are identified (see `<sys/socket.h>`), of which only the UNIX and Internet domains are normally used Solaris 2.x Sockets can be used to communicate between processes on a single system, like other forms of IPC.

The UNIX domain provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. Sockets can also be used to communicate between processes on different systems. The socket address space between connected systems is called the Internet domain.

Internet domain communication uses the TCP/IP internet protocol suite.

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type. There are five types of socket.

A stream socket — provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries. A stream operates much like a telephone conversation. The socket type is `SOCK_STREAM`, which, in the Internet domain, uses Transmission Control Protocol (TCP).

A datagram socket — supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent. Record boundaries in the data are preserved. Datagram sockets operate much like passing letters back and forth in the mail. The socket type is `SOCK_DGRAM`, which, in the Internet domain, uses User Datagram Protocol (UDP).

A sequential packet socket — provides a two-way, sequenced, reliable, connection, for datagrams of a fixed maximum length. The socket type is `SOCK_SEQPACKET`. No protocol for this type has been implemented for any protocol family.

A raw socket provides access to the underlying communication protocols.

These sockets are usually datagram oriented, but their exact characteristics depend on the interface provided by the protocol.

27.1 Socket Creation and Naming

`int socket(int domain, int type, int protocol)` is called to create a socket in the specified domain and of the specified type. If a `protocol` is not specified, the system defaults to a protocol that supports the specified socket type. The socket handle (a descriptor) is returned. A remote process has no way to identify a socket until an address is bound to it. Communicating processes connect through addresses. In the UNIX domain, a connection is usually composed of one or two path names. In the Internet domain, a connection is composed of local and remote addresses and local and remote ports. In most domains, connections must be unique.

`int bind(int s, const struct sockaddr *name, int namelen)` is called to bind a path or internet address to a socket. There are three different ways to call `bind()`, depending on the domain of the socket.

- For UNIX domain sockets with paths containing 14, or fewer characters, you can:

  ```
  #include <sys/socket.h>
   ...
  bind (sd, (struct sockaddr *) &addr, length);
  ```

- If the path of a UNIX domain socket requires more characters, use:

  ```
  #include <sys/un.h>
  ...
  bind (sd, (struct sockaddr_un *) &addr, length);
  ```

- For Internet domain sockets, use

  ```
  #include <netinet/in.h>
  ...
  bind (sd, (struct sockaddr_in *) &addr, length);
  ```

In the UNIX domain, binding a name creates a named socket in the file system. Use `unlink()` or `rm ()` to remove the socket.

27.2 Connecting Stream Sockets

Connecting sockets is usually not symmetric. One process usually acts as a server and the other process is the client. The server binds its socket to a previously agreed path or address. It then blocks on the socket. For a `SOCK_STREAM` socket, the server calls `int listen(int s, int backlog)` , which specifies how many connection requests can be queued. A client initiates a connection to the server's socket by a call to `int connect(int s, struct sockaddr *name, int namelen)` . A UNIX domain call is like this:

```
struct sockaddr_un server;
...
connect (sd, (struct sockaddr_un *)&server, length);
```

while an Internet domain call would be:

```
struct sockaddr_in;
...
connect (sd, (struct sockaddr_in *)&server, length);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. For a `SOCK_STREAM` socket, the server calls accept(3N) to complete the connection.

`int accept(int s, struct sockaddr *addr, int *addrlen)` returns a new socket descriptor which is valid only for the particular connection. A server can have multiple `SOCK_STREAM` connections active at one time.

27.3 Stream Data Transfer and Closing

Several functions to send and receive data from a `SOCK_STREAM` socket. These are `write()`, `read()`, `int send(int s, const char *msg, int len, int flags)`, and `int recv(int s, char *buf, int len, int flags)`. `send()` and `recv()` are very similar to `read()` and `write()`, but have some additional operational `flags`.

The flags parameter is formed from the bitwise OR of zero or more of the following:

`MSG_OOB` — Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support "out-of-band" data. Only `SOCK_STREAM` sockets created in the `AF_INET` address family support out-of-band data.

`MSG_DONTROUTE` — The `SO_DONTROUTE` option is turned on for the duration of the operation. It is used only by diagnostic or routing pro- grams.

`MSG_PEEK` — "Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

A `SOCK_STREAM` socket is discarded by calling `close()`.

27.4 Datagram sockets

A datagram socket does not require that a connection be established. Each message carries the destination address. If a particular local address is needed, a call to `bind()` must precede any data transfer. Data is sent through calls to `sendto()` or `sendmsg()`. The `sendto()` call is like a `send()` call with the destination address also specified. To receive datagram socket messages, call `recvfrom()` or `recvmsg()`. While `recv()` requires one buffer for the arriving data, `recvfrom()` requires two buffers, one for the incoming message and another to receive the source address.

Datagram sockets can also use `connect()` to connect the socket to a specified destination socket. When this is done, `send()` and `recv()` are used to send and receive data.

`accept()` and `listen()` are not used with datagram sockets.

27.5 Socket Options

Sockets have a number of options that can be fetched with `getsockopt()` and set with `setsockopt()`. These functions can be used at the native socket level (`level = SOL_SOCKET`), in which case the socket option name must be specified. To manipulate options at any other level the protocol number of the desired protocol controlling the option of interest must be specified (see `getprotoent()` in `getprotobyname()`).

27.6 Example Socket Programs:`socket_server.c`,`socket_cli`

These two programs show how you can establish a socket connection using the above functions.

27.6.1 `socket_server.c`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NSTRS       3          /* no. of strings  */
#define ADDRESS     "mysocket"  /* addr to connect */

/*
 * Strings we send to the client.
 */
char *strs[NSTRS] = {
    "This is the first string from the server.\n",
    "This is the second string from the server.\n",
    "This is the third string from the server.\n"
};

main()
{
    char c;
    FILE *fp;
    int fromlen;
    register int i, s, ns, len;
    struct sockaddr_un saun, fsaun;

    /*
     * Get a socket to work with.  This socket will
     * be in the UNIX domain, and will be a
     * stream socket.
     */
    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror("server: socket");
        exit(1);
    }

    /*
     * Create the address we will be binding to.
     */
    saun.sun_family = AF_UNIX;
    strcpy(saun.sun_path, ADDRESS);

    /*
     * Try to bind the address to the socket.  We
     * unlink the name first so that the bind won't
     * fail.
     *
     * The third argument indicates the "length" of
     * the structure, not just the length of the
     * socket name.
     */
    unlink(ADDRESS);
    len = sizeof(saun.sun_family) + strlen(saun.sun_path);

    if (bind(s, &saun, len) < 0) {
        perror("server: bind");
        exit(1);
    }

    /*
     * Listen on the socket.
     */
    if (listen(s, 5) < 0) {
        perror("server: listen");
        exit(1);
    }

    /*
     * Accept connections.  When we accept one, ns
     * will be connected to the client.  fsaun will
     * contain the address of the client.
     */
    if ((ns = accept(s, &fsaun, &fromlen)) < 0) {
        perror("server: accept");
        exit(1);
    }

    /*
     * We'll use stdio for reading the socket.
     */
    fp = fdopen(ns, "r");

    /*
     * First we send some strings to the client.
     */
    for (i = 0; i < NSTRS; i++)
        send(ns, strs[i], strlen(strs[i]), 0);

    /*
     * Then we read some strings from the client and
     * print them out.
     */
    for (i = 0; i < NSTRS; i++) {
        while ((c = fgetc(fp)) != EOF) {
            putchar(c);

            if (c == '\n')
                break;
        }
    }

    /*
     * We can simply use close() to terminate the
     * connection, since we're done with both sides.
     */
    close(s);

    exit(0);
}
```

27.6.2 socket_client.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NSTRS       3           /* no. of strings  */
#define ADDRESS     "mysocket"  /* addr to connect */

/*
 * Strings we send to the server.
 */
char *strs[NSTRS] = {
    "This is the first string from the client.\n",
    "This is the second string from the client.\n",
    "This is the third string from the client.\n"
```

```
};

main()
{
    char c;
    FILE *fp;
    register int i, s, len;
    struct sockaddr_un saun;

    /*
     * Get a socket to work with.  This socket will
     * be in the UNIX domain, and will be a
     * stream socket.
     */
    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
        perror("client: socket");
        exit(1);
    }

    /*
     * Create the address we will be connecting to.
     */
    saun.sun_family = AF_UNIX;
    strcpy(saun.sun_path, ADDRESS);

    /*
     * Try to connect to the address.  For this to
     * succeed, the server must already have bound
     * this address, and must have issued a listen()
     * request.
     *
     * The third argument indicates the "length" of
     * the structure, not just the length of the
     * socket name.
     */
    len = sizeof(saun.sun_family) + strlen(saun.sun_path);

    if (connect(s, &saun, len) < 0) {
        perror("client: connect");
        exit(1);
    }

    /*
     * We’ll use stdio for reading
     * the socket.
     */
    fp = fdopen(s, "r");

    /*
     * First we read some strings from the server
     * and print them out.
     */
    for (i = 0; i < NSTRS; i++) {
        while ((c = fgetc(fp)) != EOF) {
            putchar(c);

            if (c == ’\n’)
                break;
        }
    }

    /*
     * Now we send some strings to the server.
     */
    for (i = 0; i < NSTRS; i++)
        send(s, strs[i], strlen(strs[i]), 0);

    /*
     * We can simply use close() to terminate the
     * connection, since we’re done with both sides.
     */
    close(s);

    exit(0);
}
```

27.7 Exercises

Exercise 27.1 *Configure the above* `socket_server.c` *and* `socket_client.c` *programs for you system and compile and run them. You will need to set up socket* `ADDRESS` *definition.*

Chapter 28

Threads: Basic Theory and Libraries

This chapter examines aspects of threads and multiprocessing (and multi-threading). We will firts study a little theory of threads and also look at how threading can be effectively used to make programs more efficient. The C thread libraries will then be introduced. The following chapters will look at further thread issues such as synchronisation and practical examples.

28.1 Processes and Threads

We can think of a **thread** as basically a *lightweight* process. In order to understand this let us consider the two main characteristics of a process:

Unit of resource ownership — A process is allocated:

- a virtual address space to hold the process image
- control of some resources (files, I/O devices...)

Unit of dispatching - A process is an execution path through one or more programs:

- execution may be interleaved with other processes
- the process has an execution state and a dispatching priority

If we treat these two characteristics as being independent (as does modern OS theory):

- The unit of resource ownership is usually referred to as a **process** or task. This Processes have:
  - a virtual address space which holds the process image.
  - protected access to processors, other processes, files, and I/O resources.
- The unit of dispatching is usually referred to a **thread** or a lightweight process. Thus a thread:
  - Has an execution state (running, ready, etc.)
  - Saves thread context when not running
  - Has an execution stack and some per-thread static storage for local variables
  - Has access to the memory address space and resources of its process
- all threads of a process share this when one thread alters a (non-private) memory item, all other threads (of the process) sees that a file open with one thread, is available to others

28.1.1 Benefits of Threads vs Processes

If implemented correctly then threads have some advantages of (multi) processes, They take:

- Less time to create a new thread than a process, because the newly created thread uses the current process address space.
- Less time to terminate a thread than a process.
- Less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
- Less communication overheads — communicating between the threads of one process is simple because the threads share everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads.

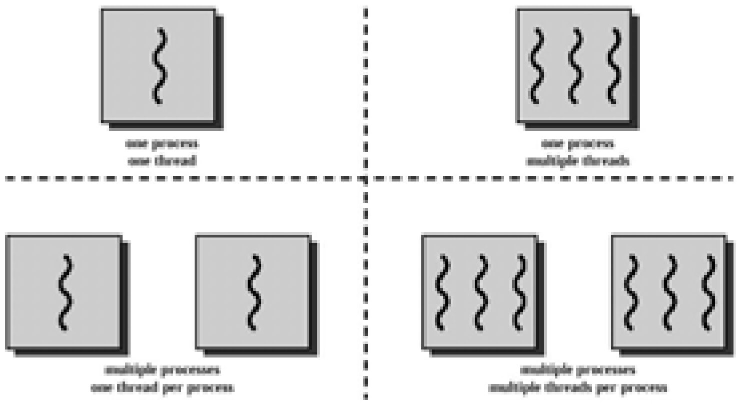


Figure 28.1: Threads and Processes

28.1.2 Multithreading vs. Single threading

Just a we can multiple processes running on some systems we can have multiple threads running:

Single threading — when the OS does not recognize the concept of thread

Multithreading — when the OS supports multiple threads of execution within a single process

Figure 28.1 shows a variety of models for threads and processes.
Some example popular OSs and their thread support is:

MS-DOS — support a single user process and a single thread

UNIX — supports multiple user processes but only supports one thread per process

Solaris — supports multiple threads

Multithreading your code can have many benefits:

- Improve application responsiveness — Any program in which many activities are not dependent upon each other can be redesigned so that each activity is defined as a thread. For example, the user of a multithreaded GUI does not have to wait for one activity to complete before starting another.

- Use multiprocessors more efficiently — Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors. Numerical algorithms and applications with a high degree of parallelism, such as matrix multiplications, can run much faster when implemented with threads on a multiprocessor.

- Improve program structure — Many programs are more efficiently structured as multiple independent or semi-independent units of execution instead of as a single, monolithic thread. Multithreaded programs can be more adaptive to variations in user demands than single threaded programs.

- Use fewer system resources — Programs that use two or more processes that access common data through shared memory are applying more than one thread of control. However, each process has a full address space and operating systems state. The cost of creating and maintaining this large amount of state information makes each process much more expensive than a thread in both time and space. In addition, the inherent separation between processes can require a major effort by the programmer to communicate between the threads in different processes, or to synchronize their actions.

Figure 28.2 illustrates different process models and thread control in a single thread and multithreaded application.

28.1.3 Some Example applications of threads

:

Example : A file server on a LAN

- It needs to handle several file requests over a short period

- Hence more efficient to create (and destroy) a single thread for each request

- Multiple threads can possibly be executing simultaneously on different processors

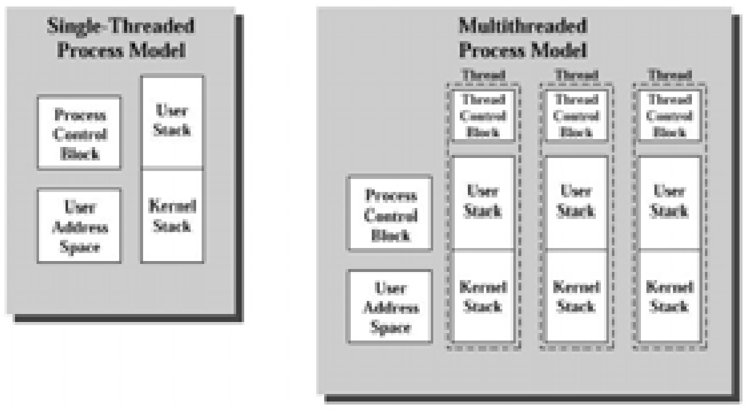


Figure 28.2: Single and Multi- Thread Applicatiions

Example 2: Matrix Multiplication

Matrix Multilication essentially involves taking the rows of one matrix and multiplying and adding corresponding columns in a second matrix *i.e*:

Note that each *element* of the resultant matrix can be computed independently, that is to say by a different thread.

We will develop a C++ example program for matrix multiplication later (see Chapter 60).

28.2 Thread Levels

There are two broad categories of thread implementation:

- User-Level Threads — Thread Libraries.
- Kernel-level Threads — System Calls.

There are merits to both, in fact some OSs allow access to both levels (*e.g.* Solaris).

28.2.1 User-Level Threads (ULT)

In this level, the kernel is not aware of the existence of threads — All thread management is done by the application by using a thread library. Thread

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

Figure 28.3: Matrix Multiplication (3x3 example)

switching does not require kernel mode privileges (no mode switch) and scheduling is application specific

Kernel activity for ULTs:

- The kernel is not aware of thread activity but it is still managing process activity
- When a thread makes a system call, the whole process will be blocked but for the thread library that thread is still in the running state
- So thread states are independent of process states

Advantages and inconveniences of ULT
Advantages:

- Thread switching does not involve the kernel — no mode switching
- Scheduling can be application specific — choose the best algorithm.
- ULTs can run on any OS — Only needs a thread library

Disadvantages:

- Most system calls are blocking and the kernel blocks processes — So all threads within the process will be blocked
- The kernel can only assign processes to processors — Two threads within the same process cannot run simultaneously on two processors

28.2.2 Kernel-Level Threads (KLT)

In this level, All thread management is done by kernel No thread library but an API (system calls) to the kernel thread facility exists. The kernel maintains context information for the process and the threads, switching between threads requires the kernel Scheduling is performed on a thread basis.

Advantages and inconveniences of KLT
Advantages

- the kernel can simultaneously schedule many threads of the same process on many processors blocking is done on a thread level
- kernel routines can be multithreaded

Disadvantages:

- thread switching within the same process involves the kernel, *e.g* if we have 2 mode switches per thread switch this results in a significant slow down.

28.2.3 Combined ULT/KLT Approaches

Idea is to combine the best of both approaches

Solaris is an example of an OS that combines both ULT and KLT (Figure 28.4:

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs

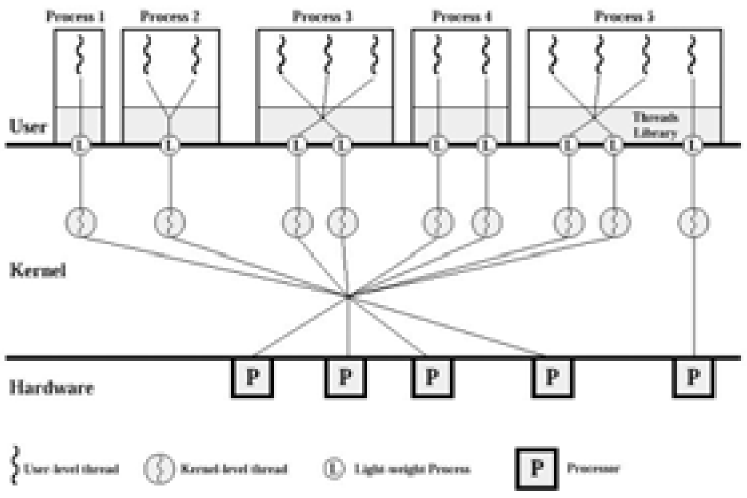


Figure 28.4: Solaris Thread Implementation

- Process includes the user's address space, stack, and process control block
- User-level threads (threads library) invisible to the OS are the interface for application parallelism
- Kernel threads the unit that can be dispatched on a processor
- Lightweight processes (LWP) each LWP supports one or more ULTs and maps to exactly one KLT

28.3 Threads libraries

The interface to multithreading support is through a subroutine library, libpthread for POSIX threads, and libthread for Solaris threads. They both contain code for:

- creating and destroying threads
- passing messages and data between threads
- scheduling thread execution
- saving and restoring thread contexts

28.4 The POSIX Threads Library:`libpthread`, <pthread.h>

28.4.1 Creating a (Default) Thread

Use the function `pthread_create()` to add a new thread of control to the current process. It is prototyped by:

```
int pthread_create(pthread\_t *tid, const pthread\_attr\_t *tattr,
void*(*start_routine)(void *), void *arg);
```

When an attribute object is not specified, it is NULL, and the *default* thread is created with the following attributes:

- It is unbounded
- It is nondetached
- It has a a default stack and stack size
- It inhetits the parent's priority

You can also create a default attribute object with `pthread_attr_init()` function, and then use this attribute object to create a default thread. See the Section 29.2.

An example call of default thread creation is:

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;
/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The `pthread_create()` function is called with `attr` having the necessary state behavior. `start_routine` is the function with which the new thread begins execution. When `start_routine` returns, the thread exits with the exit status set to the value returned by `start_routine`.

When `pthread_create` is successful, the ID of the thread created is stored in the location referred to as `tid`.

Creating a thread using a NULL attribute argument has the same effect as using a default attribute; both create a default thread. When tattr is initialized, it acquires the default behavior.

`pthread_create()` returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred.

28.4.2 Wait for Thread Termination

Use the `pthread_join` function to wait for a thread to terminate. It is prototyped by:

```
int pthread_join(thread_t tid, void **status);
```

An example use of this function is:

```
#include <pthread.h>
pthread_t tid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);
/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

The `pthread_join()` function blocks the calling thread until the specified thread terminates. The specified thread must be in the current process and must not be detached. When `status` is not `NULL`, it points to a location that is set to the exit status of the terminated thread when `pthread_join()` returns successfully. Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of `ESRCH`. After `pthread_join()` returns, any stack storage associated with the thread can be reclaimed by the application.

The `pthread_join()` routine takes two arguments, giving you some flexibility in its use. When you want the caller to wait until a specific thread terminates, supply that thread's ID as the first argument. If you are interested in the exit code of the defunct thread, supply the address of an area to receive it. Remember that `pthread_join()` works only for target threads that are nondetached. When there is no reason to synchronize with the termination of a particular thread, then that thread should be detached. Think of a detached thread as being the thread you use in most instances and reserve nondetached threads for only those situations that require them.

28.4.3 A Simple Threads Example

In this Simple Threads fragment below, one thread executes the procedure at the top, creating a helper thread that executes the procedure fetch, which involves a complicated database lookup and might take some time.

The main thread wants the results of the lookup but has other work to do in the meantime. So it does those other things and then waits for its helper to complete its job by executing `pthread_join()`. An argument, `pbe`, to the new thread is passed as a stack parameter. This can be done here because the main thread waits for the spun-off thread to terminate. In general, though, it is better to `malloc()` storage from the heap instead of passing an address to thread stack storage, which can disappear or be reassigned if the thread terminated.

The source for `thread.c` is as follows:

```
void mainline (...)
{
struct phonebookentry *pbe;
pthread_attr_t tattr;
pthread_t helper;
int status;
pthread_create(&helper, NULL, fetch, &pbe);
/* do something else for a while */
pthread_join(helper, &status);
/* it's now safe to use result */
}
void fetch(struct phonebookentry *arg)
{
```

```
struct phonebookentry *npbe;
/* fetch value from a database */
npbe = search (prog_name)
if (npbe != NULL)
*arg = *npbe;
pthread_exit(0);
}
struct phonebookentry {
char name[64];
char phonenumber[32];
char flags[16];
}
```

28.4.4 Detaching a Thread

The function `pthread_detach()` is an alternative to `pthread_join()` to reclaim storage for a thread that is created with a detachstate attribute set to `PTHREAD_CREATE_JOINABLE`. It is prototyped by:

```
int pthread\_detach(thread\_t tid);
```

A simple example of calling this fucntion to detatch a thread is given by:

```
#include <pthread.h>
pthread_t tid;
int ret;
/* detach thread tid */
ret = pthread_detach(tid);
```

The `pthread_detach()` function is used to indicate to the implementation that storage for the thread `tid` can be reclaimed when the thread terminates. If tid has not terminated, `pthread_detach()` does not cause it to terminate. The effect of multiple `pthread_detach()` calls on the same target thread is unspecified.

`pthread_detach()` returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `pthread_detach()` fails and returns the an error value.

28.4.5 Create a Key for Thread-Specific Data

Single-threaded C programs have two basic classes of data: local data and global data. For multithreaded C programs a third class is added:*thread-specific data (TSD).* This is very much like global data, except that it is private to a thread.

Thread-specific data is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a key that is global to all threads in the process. Using the key, a thread can access a pointer `(void *)` that is maintained per-thread.

The function `pthread_keycreate()` is used to allocate a key that is used to identify thread-specific data in a process. The key is global to all threads in the process, and all threads initially have the value `NULL` associated with the key when it is created.

`pthread_keycreate()` is called once for each key before the key is used. There is no implicit synchronization. Once a key has been created, each thread can bind a value to the key. The values are specific to the thread and are maintained for each thread independently. The per-thread binding is deallocated when a thread terminates if the key was created with a destructor function. `pthread_keycreate()` is prototyped by:

```
int pthread_key_create(pthread_key_t *key, void (*destructor) (void *));
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
int ret;
/* key create without destructor */
ret = pthread_key_create(&key, NULL);
/* key create with destructor */
ret = pthread_key_create(&key, destructor);
```

When `pthread_keycreate()` returns successfully, the allocated `key` is stored in the location pointed to by `key`. The caller must ensure that the storage and access to this key are properly synchronized. An optional destructor function, destructor, can be used to free stale storage. When a key has a non-NULL destructor function and the thread has a non-`NULL` value associated with that key, the destructor function is called with the current associated value when the thread exits. The order in which the destructor functions are called is unspecified.

`pthread_keycreate()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `pthread_keycreate()` fails and returns an error value.

28.4.6 Delete the Thread-Specific Data Key

The function `pthread_keydelete()` is used to destroy an existing thread-specific data key. Any memory associated with the key can be freed because the key has been invalidated and will return an error if ever referenced. (There is no comparable function in Solaris threads.)

`pthread_keydelete()` is prototyped by:

```
int pthread_key_delete(pthread_key_t key);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
int ret;
/* key previously created */
ret = pthread_key_delete(key);
```

Once a `key` has been deleted, any reference to it with the `pthread_setspecific()` or `pthread_getspecific()` call results in the `EINVAL` error.

It is the responsibility of the programmer to free any thread-specific resources before calling the delete function. This function does not invoke any of the destructors.

`pthread_keydelete()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_keycreate()` fails and returns the corresponding value.

28.4.7 Set the Thread-Specific Data Key

The function `pthread_setspecific()` is used to set the thread-specific binding to the specified thread-specific data key. It is prototyped by :

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
void *value;
int ret;

/* key previously created */
ret = pthread_setspecific(key, value);
```

`pthread_setspecific()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `pthread_setspecific()` fails and returns an error value.

Note: `pthread_setspecific()` does *not* free its storage. If a new binding is set, the existing binding must be freed; otherwise, a *memory leak can occur.*

28.4.8 Get the Thread-Specific Data Key

Use `pthread_getspecific()` to get the calling thread's binding for key, and store it in the location pointed to by value. This function is prototyped by:

```
int pthread_getspecific(pthread_key_t key);
```

A simple example use of this function is:

```
#include <pthread.h>
pthread_key_t key;
void *value;
/* key previously created */
value = pthread_getspecific(key);
```

28.4.9 Global and Private Thread-Specific Data Example

Thread-Specific Data Global but Private
Consider the following code:

```
body() {
...
while (write(fd, buffer, size) == -1) {
if (errno != EINTR) {
fprintf(mywindow, "%s\n", strerror(errno));
exit(1);
}
}
...
}
```

This code may be executed by any number of threads, but it has references to two global variables, errno and mywindow, that really should be references to items private to each thread.

References to errno should get the system error code from the routine called by this thread, not by some other thread. So, references to errno by one thread refer to a different storage location than references to errno by other threads. The mywindow variable is intended to refer to a stdio stream connected to a window that is private to the referring thread. So, as with `errno`, references to mywindow by one thread should refer to a different storage location (and, ultimately, a different window) than references to mywindow by other threads. The only difference here is that the threads library takes care of errno, but the programmer must somehow make this work for mywindow. The next example shows how the references to mywindow work. The preprocessor converts references to mywindow into invocations of the `mywindow` procedure. This routine in turn invokes `pthread_getspecific()`, passing it the `mywindow_key` global variable (it really is a global variable) and an output parameter, `win`, that receives the identity of this thread's window.

Turning Global References Into Private References Now consider this code fragment:

```
thread_key_t mywin_key;
```

```
FILE *_mywindow(void) {
FILE *win;
pthread_getspecific(mywin_key, &win);
return(win);
}
#define mywindow _mywindow()

void routine_uses_win( FILE *win) {
...
}
void thread_start(...) {
...
make_mywin();
...
routine_uses_win( mywindow )
...
}
```

The `mywin_key` variable identifies a class of variables for which each thread has its own private copy; that is, these variables are thread-specific data. Each thread calls `make_mywin` to initialize its window and to arrange for its instance of mywindow to refer to it. Once this routine is called, the thread can safely refer to `mywindow` and, after `mywindow`, the thread gets the reference to its private window. So, references to mywindow behave as if they were direct references to data private to the thread.

We can now set up our initial Thread-Specific Data:

```
void make_mywindow(void) {
FILE **win;
static pthread_once_t mykeycreated = PTHREAD_ONCE_INIT;
pthread_once(&mykeycreated, mykeycreate);
win = malloc(sizeof(*win));
create_window(win, ...);
pthread_setspecific(mywindow_key, win);
}
void mykeycreate(void) {
pthread_keycreate(&mywindow_key, free_key);
}
void free_key(void *win) {
free(win);
}
```

First, get a unique value for the key, `mywin_key`. This key is used to identify the thread-specific class of data. So, the first thread to call `make_mywin` eventually calls `pthread_keycreate()`, which assigns to its first argument a unique key. The second argument is a destructor function that is used to deallocate a thread's instance of this thread-specific data item once the thread terminates.

The next step is to allocate the storage for the caller's instance of this thread-specific data item. Having allocated the storage, a call is made to the `create_window` routine, which sets up a window for the thread and sets the storage pointed to by win to refer to it. Finally, a call is made to `pthread_setspecific()`, which associates the value contained in win (that is, the location of the storage containing the reference to the window) with the key. After this, whenever this thread calls `pthread_getspecific()`, passing the global key, it gets the value that was associated with this key by this thread when it called `pthread_setspecific()`. When a thread terminates, calls are made to the destructor functions that were set up in `pthread_key_create()`. Each destructor function is called only if the terminating thread established a value for the key by calling `pthread_setspecific()`.

28.4.10 Getting the Thread Identifiers

The function `pthread_self()` can be called to return the ID of the calling thread. It is prototyped by:

```
pthread_t pthread_self(void);
```

It is use is very straightforward:

```
#include <pthread.h>
pthread_t tid;
tid = pthread_self();
```

28.4.11 Comparing Thread IDs

The function `pthread_equal()` can be called to compare the thread identification numbers of two threads. It is prototyped by:

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

It is use is straightforward to use, also:

```
#include <pthread.h>
pthread_t tid1, tid2;
int ret;
ret = pthread_equal(tid1, tid2);
```

As with other comparison functions, `pthread_equal()` returns a non-zero value when `tid1` and `tid2` are equal; otherwise, zero is returned. When either `tid1` or `tid2` is an invalid thread identification number, the result is unpredictable.

28.4.12 Initializing Threads

Use `pthread_once()` to call an initialization routine the first time `pthread_once()` is called — Subsequent calls to have no effect. The prototype of this function is:

```
int pthread_once(pthread_once_t *once_control,
void (*init_routine)(void));
```

28.4.13 Yield Thread Execution

The function `sched_yield()` to cause the current thread to yield its execution in favor of another thread with the same or greater priority. It is prototyped by:

```
int sched_yield(void);
```

It is clearly a simple function to call:

```
#include <sched.h>
int ret;
ret = sched_yield();
```

`sched_yield()` returns zero after completing successfully. Otherwise -1 is returned and errno is set to indicate the error condition.

28.4.14 Set the Thread Priority

Use `pthread_setschedparam()` to modify the priority of an existing thread. This function has no effect on scheduling policy. It is prototyped as follows:

```
int pthread_setschedparam(pthread_t tid, int policy,
const struct sched_param *param);
```

and used as follows:

```
#include <pthread.h>
pthread_t tid;
int ret;
struct sched_param param;
int priority;
/* sched_priority will be the priority of the thread */
sched_param.sched_priority = priority;
/* only supported policy, others will result in ENOTSUP */

policy = SCHED_OTHER;
/* scheduling parameters of target thread */
ret = pthread_setschedparam(tid, policy, &param);
```

`pthread_setschedparam()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the `pthread_setschedparam()` function fails and returns an error value.

28.4.15 Get the Thread Priority

`int pthread_getschedparam(pthread_t tid, int policy, struct schedparam *param)` gets the priority of the existing thread.

An example call of this function is:

```
#include <pthread.h>
pthread_t tid;
sched_param param;
int priority;
int policy;
int ret;
/* scheduling parameters of target thread */
ret = pthread_getschedparam (tid, &policy, &param);
/* sched_priority contains the priority of the thread */
priority = param.sched_priority;
```

`pthread_getschedparam()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the error value set.

28.4.16 Send a Signal to a Thread

Signal may be sent to threads is a similar fashion to those for process as follows:

```
#include <pthread.h>
#include <signal.h>
int sig;
pthread_t tid;
int ret;
ret = pthread_kill(tid, sig);
```

`pthread_kill()` sends the signal `sig` to the thread specified by `tid`. `tid` must be a thread within the same process as the calling thread. The `sig` argument must be a valid signal of the same type defined for `signal()` in <`signal.h`> (See Chapter 23)

When `sig` is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of tid.

This function returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, `pthread_kill()` fails and returns an error value.

28.4.17 Access the Signal Mask of the Calling Thread

The function `pthread_sigmask()` may be used to change or examine the signal mask of the calling thread. It is prototyped as follows:

```
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);
```

Example uses of this function include:

```
#include <pthread.h>
#include <signal.h>
int ret;
sigset_t old, new;
ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* set new mask */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

how determines how the signal set is changed. It can have one of the following values:

`SIG_SETMASK` — Replace the current signal mask with new, where new indicates the new signal mask.

`SIG_BLOCK` — Add new to the current signal mask, where new indicates the set of signals to block.

`SIG_UNBLOCK` — Delete new from the current signal mask, where new indicates the set of signals to unblock.

When the value of `new` is `NULL`, the value of how is not significant and the signal mask of the thread is unchanged. So, to inquire about currently blocked signals, assign a `NULL` value to the new argument. The old variable points to the space where the previous signal mask is stored, unless it is `NULL`.

`pthread_sigmask()` returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_sigmask()` fails and returns an errro value.

28.4.18 Terminate a Thread

A thread can terminate its execution in the following ways:

- By returning from its first (outermost) procedure, the threads start routine; see `pthread_create()`
- By calling `pthread_exit()`, supplying an exit status

- By termination with POSIX cancel functions; see `pthread_cancel()`

The `void pthread_exit(void *status)` is used terminate a thread in a similar fashion the `exit()` for a process:

```
#include <pthread.h>
int status;
pthread_exit(&status); /* exit with status */
```

The `pthread_exit()` function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by status are retained until the thread is waited for (blocked). Otherwise, status is ignored and the thread's ID can be reclaimed immediately.

The `pthread_cancel()` function to cancel a thread is prototyped:

```
int pthread_cancel(pthread_t thread);
```

and called:

```
#include <pthread.h>
pthread_t thread;
int ret;
ret = pthread_cancel(thread);
```

How the cancellation request is treated depends on the state of the target thread. Two functions,

`pthread_setcancelstate()` and `pthread_setcanceltype()` (see `man` pages for further information on these functions), determine that state.

`pthread_cancel()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns an error value.

28.5 Solaris Threads: <`thread.h`>

Solaris have many similarities to POSIX threads,In this sectionfocus on the Solaris features that are not found in POSIX threads. Where functionality is virtually the same for both Solaris threads and for pthreads, (even though the function names or arguments might differ), only a brief example consisting of the correct include file and the function prototype is presented. Where return values are not given for the Solaris threads functions, see the appropriate `man` pages.

The Solaris threads API and the pthreads API are two solutions to the same problem: building parallelism into application software. Although each API is complete in itself, you can safely mix Solaris threads functions and pthread functions in the same program.

The two APIs do not match exactly, however. Solaris threads supports functions that are not found in pthreads, and pthreads includes functions that are not supported in the Solaris interface. For those functions that do match, the associated arguments might not, although the information content is effectively the same.

By combining the two APIs, you can use features not found in one to enhance the other. Similarly, you can run applications using Solaris threads, exclusively, with applications using pthreads, exclusively, on the same system.

To use the Solaris threads functions described in this chapter, you must link with the Solaris threads library `-lthread` and include the <`thread.h`> in all programs.

28.5.1 Unique Solaris Threads Functions

Let us begin by looking at some functions that are unique to Solaris threads:

- Suspend Thread Execution
- Continue a Suspended Thread
- Set Thread Concurrency Level
- Get Thread Concurrency

Suspend Thread Execution

The function `thr_suspend()` immediately suspends the execution of the thread specified by a target thread, (`tid` below). It is prototyped by:

```
int thr_suspend(thread_t tid);
```

On successful return from `thr_suspend()`, the suspended thread is no longer executing. Once a thread is suspended, subsequent calls to `thr_suspend()` have no effect. Signals cannot awaken the suspended thread; they remain pending until the thread resumes execution.

A simple example call is as follows:

```
#include <thread.h>

thread_t tid; /* tid from thr_create() */
/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create() */
pthread_t ptid;
int ret;
ret = thr_suspend(tid);
/* using pthreads ID variable with a cast */
ret = thr_suspend((thread_t) ptid);
```

Note: `pthread_t tid` as defined in pthreads is the same as `thread_t` tid in Solaris threads. `tid` values can be used interchangeably either by assignment or through the use of casts.

Continue a Suspended Thread

The function `thr_continue()` resumes the execution of a suspended thread. It is prototypes as follows:

```
int thr_continue(thread_t tid);
```

Once a suspended thread is continued, subsequent calls to `thr_continue()` have no effect.

A suspended thread will *not* be awakened by a signal. The signal stays pending until the execution of the thread is resumed by `thr_continue()`.

`thr_continue()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_continue()` The following code fragment illustrates the use of the function:

```
thread_t tid; /* tid from thr_create()*/
/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create()*/
pthread_t ptid;
int ret;
ret = thr_continue(tid);
/* using pthreads ID variable with a cast */
ret = thr_continue((thread_t) ptid)
```

Set Thread Concurrency Level

By default, Solaris threads attempt to adjust the system execution resources (LWPs) used to run unbound threads to match the real number of active threads. While the Solaris threads package cannot make perfect decisions, it at least ensures that the process continues to make progress. When you have some idea of the number of unbound threads that should be simultaneously active (executing code or system calls), tell the library through `thr_setconcurrency(int new_level)`. To get the number of threads being used, use the function `thr_getconcurrencyint(void)`:

`thr_setconcurrency()` provides a hint to the system about the required level of concurrency in the application. The system ensures that a sufficient number of threads are active so that the process continues to make progress, for example:

```
#include <thread.h>
int new_level;
int ret;

ret = thr_setconcurrency(new_level);
```

Unbound threads in a process might or might not be required to be simultaneously active. To conserve system resources, the threads system ensures by default that enough threads are active for the process to make progress, and that the process will not deadlock through a lack of concurrency. Because this might not produce the most effective level of concurrency, `thr_setconcurrency()` permits the application to give the threads system a hint, specified by `new_level`, for the desired level of concurrency. The actual number of simultaneously active threads can be larger or smaller than `new_level`. Note that an application with multiple compute-bound threads can fail to schedule all the runnable threads if `thr_setconcurrency()` has not been called to adjust the level of execution resources. You can also affect the value for the desired concurrency level by setting the `THR_NEW_LW` flag in `thr_create()`. This effectively increments the current level by one.

`thr_setconcurrency()` a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_setconcurrency()` fails and returns the corresponding value to ẽrrno.

Readers/Writer Locks

Readers/Writer locks are another unique feature of Solaris threads. They allow simultaneous read access by many threads while restricting write access to only one thread at a time.

When any thread holds the lock for reading, other threads can also acquire the lock for reading but must wait to acquire the lock for writing. If one thread holds the lock for writing, or is waiting to acquire the lock for writing, other threads must wait to acquire the lock for either reading or writing. Readers/writer locks are slower than mutexes, but can improve performance when they protect data that are not frequently written but that are read by many concurrent threads. Use readers/writer locks to synchronize threads in this process and other processes by allocating them in memory that is writable and shared among the cooperating processes (see mmap(2)) and by initializing them for this behavior. By default, the acquisition order is not defined when multiple threads are waiting for a readers/writer lock. However, to avoid writer starvation, the Solaris threads package tends to favor writers over readers. Readers/writer locks must be initialized before use.

Initialize a Readers/Writer Lock

The function `rwlock_init()` initialises the readers/writer lock. it is prototypes in <`synch.h`> or <`thread.h`> as follows:

```
int rwlock_init(rwlock_t *rwlp, int type, void * arg);
```

The readers/writer lock pointed to by `rwlp` and to set the lock state to unlocked. `type` can be one of the following

`USYNC_PROCESS` — The readers/writer lock can be used to synchronize threads in this process and other processes.

`USYNC_THREAD` — The readers/writer lock can be used to synchronize threads in this process, only.

Note: that `arg` is currently ignored.

`rwlock_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value to `errno`.

Multiple threads must not initialize the same readers/writer lock simultaneously. Readers/writer locks can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. A readers/writer lock must not be reinitialized while other threads might be using it.

An example code fragment that initialises Readers/Writer Locks with Intraprocess Scope is as follows:

```
#include <thread.h>

rwlock_t rwlp;
int ret;
/* to be used within this process only */
ret = rwlock_init(&rwlp, USYNC_THREAD, 0);
Initializing Readers/Writer Locks with Interprocess Scope
#include <thread.h>
rwlock_t rwlp;
int ret;
/* to be used among all processes */
ret = rwlock_init(&rwlp, USYNC_PROCESS, 0);
```

Acquire a Read Lock

To acquire a read lock on the readers/writer lock use the `rw_rdlock()` function:

```
int rw_rdlock(rwlock_t *rwlp);
```

The readers/writer lock pointed to by `rwlp`. When the readers/writer lock is already locked for writing, the calling thread blocks until the write lock is released. Otherwise, the read lock is acquired.

`rw_rdlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value to `errno`.

A function `rw_tryrdlock(rwlock_t *rwlp)` may also be used to attempt to acquire a read lock on the readers/writer lock pointed to by rwlp. When the readers/writer lock is already locked for writing, it returns an error. Otherwise, the read lock is acquired. This function returns zero after completing successfully. Any other returned value indicates that an error occurred.

Acquire a Write Lock

The function `rw_wrlock(rwlock_t *rwlp)` acquires a write lock on the readers/writer lock pointed to by `rwlp`. When the readers/writer lock is already locked for reading or writing, the calling thread blocks until all the read locks and write locks are released. Only one thread at a time can hold a write lock on a readers/writer lock.

`rw_wrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Use `rw_trywrlockrwlock_t *rwlp)` to attempt to acquire a write lock on the readers/writer lock pointed to by rwlp. When the readers/writer lock is already locked for reading or writing, it returns an error.

`rw_trywrlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Unlock a Readers/Writer Lock

The function `rw_unlock(rwlock_t *rwlp)` unlocks a readers/writer lock pointed to by `rwlp`. The readers/writer lock must be locked and the calling thread must hold the lock either for reading or writing. When any other threads are waiting for the readers/writer lock to become available, one of them is unblocked.

`rw_unlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Destroy Readers/Writer Lock State

The function `rwlock_destroy(rwlock_t *rwlp)` destroys any state associated with the readers/writer lock pointed to by `rlwp`. The space for storing the readers/writer lock is not freed.

`rwlock_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Readers/Writer Lock Example

The following example uses a bank account analogy to demonstrate readers/writer locks. While the program could allow multiple threads to have concurrent read-only access to the account balance, only a single writer is allowed. Note that the `get_balance()` function needs the lock to ensure that the addition of the checking and saving balances occurs atomically.

```
rwlock_t account_lock;
float checking_balance = 100.0;
float saving_balance = 100.0;
...
rwlock_init(&account_lock, 0, NULL);
...
float
get_balance() {
float bal;
rw_rdlock(&account_lock);
bal = checking_balance + saving_balance;
rw_unlock(&account_lock);
return(bal);
}
void
transfer_checking_to_savings(float amount) {
rw_wrlock(&account_lock);
checking_balance = checking_balance - amount;
saving_balance = saving_balance + amount;
rw_unlock(&account_lock);
}
```

28.5.2 Similar Solaris Threads Functions

Here we simply list the similar thread functions and their prototype definitions, except where the complexity of the function merits further exposition.
.

Create a Thread

The `thr_create()` routine is one of the most elaborate of all the Solaris threads library routines.

It is prototyped as follows:

```
int thr_create(void *stack_base, size_t stack_size,
void *(*start_routine) (void *), void *arg, long flags,
thread_t *new_thread);
```

Thjis function adds a new thread of control to the current process. Note that the new thread does not inherit pending signals, but it does inherit priority and signal masks.

`stack_base` contains the address for the stack that the new thread uses. If `stack_base` is NULL then `thr_create()` allocates a stack for the new thread with at least `stac_size` bytes. `stack_size` Contains the size, in number of bytes, for the stack that the new thread uses. If `stack_size` is zero, a default size is used. In most cases, a zero value works best. If `stack_size` is not zero, it must be greater than the value returned by `thr_min_stack(void)` inquiry function.

There is no general need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved.

`start_routine` contains the function with which the new thread begins execution. When `start_routine` returns, the thread exits with the exit status set to the value returned by `start_routine`

`arg` can be anything that is described by void, which is typically any 4-byte value. Anything larger must be passed indirectly by having the argument point to it.

Note that you can supply only one argument. To get your procedure to take multiple arguments, encode them as one (such as by putting them in a structure).

`flags` specifies attributes for the created thread. In most cases a zero value works best. The value in flags is constructed from the bitwise inclusive OR of the following:

`THR_SUSPENDED` — Suspends the new thread and does not execute `start_routine` until the thread is started by `thr_continue()`. Use this to operate on the thread (such as changing its priority) before you run it. The termination of a detached thread is ignored.

`THR_DETACHED` — Detaches the new thread so that its thread ID and other resources can be reused as soon as the thread terminates. Set this when you do not want to wait for the thread to terminate. Note - When there is no explicit synchronization to prevent it, an unsuspended, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `thr_create()`.

`THR_BOUND` — Permanently binds the new thread to an LWP (the new thread is a bound thread).

`THR_NEW_LWP` — Increases the concurrency level for unbound threads by one. The effect is similar to incrementing concurrency by one with `thr_setconcurrency()`, although `THR_NEW_LWP` does not affect the level set through the `thr_setconcurrency()` function. Typically, `THR_NEW_LWP` adds a new LWP to the pool of LWPs running unbound threads.

When you specify both `THR_BOUND` and `THR_NEW_LWP`, two LWPs are typically created — one for the bound thread and another for the pool of LWPs running unbound threads.

`THR_DAEMON` —- Marks the new thread as a daemon. The process exits when all nondaemon threads exit. Daemon threads do not affect the process exit status and are ignored when counting the number of thread exits.

A process can exit either by calling `exit()` or by having every thread in the process that was not created with the `THR_DAEMON` flag call `thr_exit()`. An application, or a library it calls, can create one or more threads that should be ignored (not counted) in the decision of whether to exit. The `THR_DAEMONl` flag identifies threads that are not counted in the process exit criterion.

`new_thread` points to a location (when `new_thread` is not `NULL`) where the ID of the new thread is stored when `thr_create()` is successful. The caller is responsible for supplying the storage this argument points to. The ID is valid only within the calling process. If you are not interested in this identifier, supply a zero value to `new_thread`.

`thr_create()` returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_create()` fails and returns the corresponding value to `errno`.

Get the Thread Identifier

The `int thr_self(void)` to get the ID of the calling thread.

Yield Thread Execution

`void thr_yield(void)` causes the current thread to yield its execution in favor of another thread with the same or greater priority; otherwise it has no effect. There is no guarantee that a thread calling `thr_yield()` will do so.

Signals and Solaris Threads

The following functions exist and operate as do pthreads.

`int thr_kill(thread_t target_thread, int sig)` sends a signal to a thread.

`int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset)` to change or examine the signal mask of the calling thread.

Terminating a Thread

The `void th_exit(void *status)` to terminates a thread.

The `int thr_join(thread_t tid, thread_t *departedid, void **status)` function to wait for a thread to terminate.

Therefore to join specific threads one would do:

```
#include <thread.h>
thread_t tid;
thread_t departedid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = thr_join(tid, &departedid, (void**)&status);
/* waiting to join thread "tid" without status */
ret = thr_join(tid, &departedid, NULL);
/* waiting to join thread "tid" without return id and status */
ret = thr_join(tid, NULL, NULL);
```

When the `tid` is `(thread_t) 0`, then `thread_join()` waits for any undetached thread in the process to terminate. In other words, when no thread identifier is specified, any undetached thread that exits causes `thread_join()` to return.

To join any threads:

```
#include <thread.h>
thread_t tid;
thread_t departedid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = thr_join(NULL, &departedid, (void **)&status);
```

By indicating NULL as `thread id` in the `thr_join()`, a join will take place when any non detached thread in the process exits. The departedid will indicate the thread ID of exiting thread.

`Creating a Thread-Specific Data Key`

Except for the function names and arguments, thread specific data is the same for Solaris as it is for POSIX.

`int thr_keycreate(thread_key_t *keyp, void (*destructor) (void *value))` allocates a key that is used to identify thread-specific data in a process.

`int thr_setspecific(thread_key_t key, void *value)` binds value to the thread-specific data key, key, for the calling thread.

`int thr_getspecific(thread_key_t key, void **valuep)` stores the current value bound to key for the calling thread into the location pointed to by valuep.

In Solaris threads, if a thread is to be created with a priority other than that of its parent's, it is created in `SUSPEND` mode. While suspended, the threads priority is modified using the `int thr_setprio(thread_t tid, int newprio)` function call; then it is continued.

An unbound thread is usually scheduled only with respect to other threads in the process using simple priority levels with no adjustments and no kernel involvement. Its system priority is usually uniform and is inherited from the creating process.

The function `thr_setprio()` changes the priority of the thread, specified by `tid`, within the current process to the priority specified by newprio.

By default, threads are scheduled based on fixed priorities that range from zero, the least significant, to the largest integer. The `tid` will preempt lower priority threads, and will yield to higher priority threads. For example:

```
#include <thread.h>
thread_t tid;
int ret;
int newprio = 20;
/* suspended thread creation */
ret = thr_create(NULL, NULL, func, arg, THR_SUSPEND, &tid);
/* set the new priority of suspended child thread */
ret = thr_setprio(tid, newprio);
/* suspended child thread starts executing with new priority */

ret = thr_continue(tid);
```

Use $\widehat{\text{int}}$ thr_getprio(thread_t tid, int *newprio) to get the current priority for the thread. Each thread inherits a priority from its creator. `thr_getprio()` stores the current priority, `tid`, in the location pointed to by `newprio`.

Example Use of Thread Specific Data:Rethinking Global Variables

Historically, most code has been designed for single-threaded programs. This is especially true for most of the library routines called from C programs. The following implicit assumptions were made for single-threaded code:

- When you write into a global variable and then, a moment later, read from it, what you read is exactly what you just wrote.
- This is also true for nonglobal, static storage.
- You do not need synchronization because there is nothing to synchronize with.

The next few examples discuss some of the problems that arise in multi-threaded programs because of these assumptions, and how you can deal with them.

Traditional, single-threaded C and UNIX have a convention for handling errors detected in system calls. System calls can return anything as a functional value (for example, write returns the number of bytes that were transferred). However, the value -1 is reserved to indicate that something went wrong. So, when a system call returns -1, you know that it failed.

Consider the following piece of code:

```
extern int errno;

...

if (write(file_desc, buffer, size) == -1)
  { /* the system call failed */
    fprintf(stderr, "something went wrong, error code = %d\n", errno);
    exit(1);
  }
```

Rather than return the actual error code (which could be confused with normal return values), the error code is placed into the global variable `errno`. When the system call fails, you can look in `errno` to find out what went wrong.

Now consider what happens in a multithreaded environment when two threads fail at about the same time, but with different errors.

- Both expect to find their error codes in `errno`,
- **but** one copy of errno cannot hold both values.a

This global variable approach simply does not work for multithreaded programs. Threads solves this problem through a conceptually new storage class: *thread-specific data*.

This storage is similar to global storage in that it can be accessed from any procedure in which a thread might be running. However, it is private to the thread: when two threads refer to the thread-specific data location of the same name, they are referring to two different areas of storage.

So, when using threads, each reference to `errno` is thread-specific because each thread has a private copy of `errno`. This is achieved in this implementation by making `errno` a macro that expands to a function call.

28.6 Compiling a Multithreaded Application

There are many options to consider for header files, define flags, and linking.

28.6.1 Preparing for Compilation

The following items are required to compile and link a multithreaded program.

- A standard C compiler (`cc, gcc` *etc*)
- Include files:
  - <thread.h> and <pthread.h>
  - <errno.h¿, <limits.h>, <signal.h>, <unistd.h>
- The Solaris threads library (`libthread`), the POSIX threads library (`libpthread`), and possibly the POSIX realtime library (`libposix4`) for semaphores
- MT-safe libraries (`libc, libm, libw, libintl, libnsl, libsocket, libmalloc, libmapmalloc`, and so on)

The include file <thread.h>, used with the `-lthread` library, compiles code that is upward compatible with earlier releases of the Solaris system. This library contains both interfaces: those with Solaris semantics and those with POSIX semantics. To call `thr_setconcurrency()` with POSIX threads, your program needs to include <thread.h>.

The include file <pthread.h>, used with the `-lpthread` library, compiles code that is conformant with the multithreading interfaces defined by the POSIX 1003.1c standard. For complete POSIX compliance, the define flag `_POSIX_C_SOURCE` should be set to a (long) value ≥ 199506, as follows:

```
cc [flags] file... -D_POSIX_C_SOURCE=N (where N 199506L)
```

You can mix Solaris threads and POSIX threads in the same application, by including both <thread.h> and <pthread.h>, and linking with either the `-lthread` or `-lpthread` library. In mixed use, Solaris semantics prevail when compiling with `-D_REENTRANT` flag set $\geq 199506L$ and linking with `-lthread`, whereas POSIX semantics prevail when compiling with `D_POSIX_C_SOURCE` flag set $\geq 199506L$ and linking with `-lpthread`. Defining `_REENTRANT` or `_POSIX_C_SOURCE`

Linking With libthread or libpthread

For POSIX threads behavior, load the `libpthread` library. For Solaris threads behavior, load the `libthread` library. Some POSIX programmers might want to link with `-lthread`to preserve the Solaris distinction between `fork()` and `fork1()`. All that `-lpthread` really does is to make `fork()` behave the same way as the Solaris `fork1()` call, and change the behavior of `alarm()`.

To use libthread, specify `-lthread` last on the `cc` command line.

To use libpthread, specify `-lpthread` last on the `cc` command line.

Do not link a *nonthreaded* program with `-lthread` or `-lpthread`. Doing so establishes multithreading mechanisms at link time that are initiated at run time. These *slow down* a single-threaded application, waste system resources, and produce misleading results when you debug your code.

Note: For C++ programs that use threads, use the `-mt` option, rather than `-lthread`, to compile and link your application. The `-mt` option links with `libthread` and ensures proper library linking order. (Using `-lthread` might cause your program to crash (core dump).

Linking with -lposix4 for POSIX Semaphores

The Solaris semaphore routines (see Chapter 30.3) are contained in the `libthread` library. By contrast, you link with the `-lposix4` library to get the standard POSIX semaphore routines (See Chapter 25)

28.6.2 Debugging a Multithreaded Program

The following list points out some of the more frequent oversights and errors that can cause bugs in multithreaded programs.

- Passing a pointer to the caller's stack as an argument to a new thread.
- Accessing global memory (shared changeable state) without the protection of a synchronization mechanism.
- Creating deadlocks caused by two threads trying to acquire rights to the same pair of global resources in alternate order (so that one thread controls the first resource and the other controls the second resource and neither can proceed until the other gives up).
- Trying to reacquire a lock already held (recursive deadlock).

- Creating a hidden gap in synchronization protection. This is caused when a code segment protected by a synchronization mechanism contains a call to a function that frees and then reacquires the synchronization mechanism before it returns to the caller. The result is that it appears to the caller that the global data has been protected when it actually has not.

- Mixing UNIX signals with threads — it is better to use the `sigwait()` model for handling asynchronous signals.

- Forgetting that default threads are created `PTHREAD_CREATE_JOINABLE` and must be reclaimed with `pthread_join()`. **Note**, `pthread_exit()` does not free up its storage space.

- Making deeply nested, recursive calls and using large automatic arrays can cause problems because multithreaded programs have a more limited stack size than single-threaded programs.

- Specifying an inadequate stack size, or using non-default stacks. And, note that multithreaded programs (especially those containing bugs) often behave differently in two successive runs, given identical inputs, because of differences in the thread scheduling order.

In general, multithreading bugs are statistical instead of deterministic. Tracing is usually a more effective method of finding order of execution problems than is breakpoint-based debugging.

Chapter 29

Further Threads Programming:Thread Attributes (POSIX)

The previous chapter covered the basics of threads creation using default attributes. This chapter discusses setting attributes at thread creation time.

Note that only pthreads uses attributes and cancellation, so the API covered in this chapter is for POSIX threads only. Otherwise, the functionality for Solaris threads and pthreads is largely the same.

29.1 Attributes

Attributes are a way to specify behavior that is different from the default. When a thread is created with `pthread_create()` or when a synchronization variable is initialized, an attribute object can be specified. **Note:** however that the default atributes are usually sufficient for most applications.

Impottant Note: Attributes are specified *only at thread creation time*; they **cannot** be altered while the thread is **being used**.

Thus three functions are usually called in tandem

- Thread attibute intialisation — `pthread_attr_init()` create a default `pthread_attr_t tattr`
- Thread attribute value change (unless defaults appropriate) — a variety of `pthread_attr_*()` functions are available to set individual attribute

values for the `pthread_attr_t tattr` structure. (see below).

- Thread creation — a call to `pthread_create()` with approriate attribute values set in a `pthread_attr_t tattr` structure.

The following code fragment should make this point clearer:

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* call an appropriate functions to alter a default value */
ret = pthread_attr_*(&tattr,SOME_ATRIBUTE_VALUE_PARAMETER);

/* create the thread */
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

In order to save space, code examples mainly focus on the attribute setting functions and the intializing and creation functions are ommitted. These **must** of course be present in all actual code fragtments.

An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type. Once an attribute is initialized and configured, it has process-wide scope. The suggested method for using attributes is to configure all required state specifications at one time in the early stages of program execution. The appropriate attribute object can then be referred to as needed. Using attribute objects has two primary advantages:

- First, it adds to code portability. Even though supported attributes might vary between implementations, you need not modify function calls that create thread entities because the attribute object is hidden from the interface. If the target port supports attributes that are not found in the current port, provision must be made to manage the new attributes. This is an easy porting task though, because attribute objects need only be initialized once in a well-defined location.

- Second, state specification in an application is simplified. As an example, consider that several sets of threads might exist within a process, each providing a separate service, and each with its own state requirements. At some point in the early stages of the application, a thread attribute object can be initialized for each set. All future thread creations will then refer to the attribute object initialized for that type of thread. The initialization phase is simple and localized, and any future modifications can be made quickly and reliably.

Attribute objects require attention at process exit time. When the object is initialized, memory is allocated for it. This memory must be returned to the system. The pthreads standard provides function calls to destroy attribute objects.

29.2 Initializing Thread Attributes

The function `pthread_attr_init()` is used to initialize object attributes to their default values. The storage is allocated by the thread system during execution.

The function is prototyped by:

```
int pthread_attr_init(pthread_attr_t *tattr);
```

An example call to this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* initialize an attribute to the default value */
ret = pthread_attr_init(&tattr);
```

The default values for attributes (`tattr`) are:

Attribute	Value	Result
scope	`PTHREAD_SCOPE_PROCESS`	New thread is unbound - not permanently attached to LWP.
detachstate	`PTHREAD_CREATE_JOINABLE`	Exit status and thread are preserved after the thread terminates.
stackaddr	`NULL`	New thread has system-allocated stack address.
stacksize	`1` megabyte	New thread has system-defined stack size. priority New thread inherits parent thread priority.
inheritsched	`PTHREAD_INHERIT_SCHED`	New thread inherits parent thread scheduling priority.
schedpolicy	`SCHED_OTHER`	New thread uses Solaris-defined fixed priority scheduling; threads run until preempted by a higher-priority thread or until they block or yield.

This function zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns an error value (to `errno`).

29.3 Destroying Thread Attributes

The function `pthread_attr_destroy()` is used to remove the storage allocated during initialization. The attribute object becomes invalid. It is prototyped by:

```
int pthread_attr_destroy(pthread_attr_t *tattr);
```

A sample call to this functions is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

Attribites are declared as for `pthread_attr_init()` above.

`pthread_attr_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

29.4 Thread's Detach State

When a thread is created detached (`PTHREAD_CREATE_DETACHED`), its thread ID and other resources can be reused as soon as the thread terminates.

If you do not want the calling thread to wait for the thread to terminate then call the function `pthread_attr_setdetachstate()`.

When a thread is created nondetached (`PTHREAD_CREATE_JOINABLE`), it is assumed that you will be waiting for it. That is, it is assumed that you will be executing a `pthread_join()` on the thread. Whether a thread is created detached or nondetached, the process does not exit until all threads have exited.

`pthread_attr_setdetachstate()` is prototyped by:

```
int pthread_attr_setdetachstate(pthread_attr_t *tattr,int detachstate);
```

`pthread_attr_setdetachstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

An example call to detatch a thread with this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* set the thread detach state */
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

Note - When there is no explicit synchronization to prevent it, a newly created, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `pthread_create()`. For nondetached (`PTHREAD_CREATE_JOINABLE`) threads, it is very important that some thread join with it after it terminates — otherwise the resources of that thread are not released for use by new threads. This commonly results in a memory leak. So when you do not want a thread to be joined, create it as a detached thread.

It is quite common that you will wish to create a thread which is detatched from creation. The following code illustrates how this may be achieved with the standard calls to initialise and set and then create a thread:

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The function `pthread_attr_getdetachstate()` may be used to retrieve the thread create state, which can be either detached or joined. It is prototyped by:

```
int pthread_attr_getdetachstate(const pthread_attr_t *tattr, int *detachstate);
```

`pthread_attr_getdetachstate()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to this fuction is:

```
#include <pthread.h>
pthread_attr_t tattr;
int detachstate;
int ret;

/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

29.5 Thread's Set Scope

A thread may be bound (`PTHREAD_SCOPE_SYSTEM`) or an unbound (`PTHREAD_SCOPE_PROCESS`). Both these types of types are accessible **only** within a given process.

The function `pthread_attr_setscope()` to create a bound or unbound thread. It is prototyped by:

```
int pthread_attr_setscope(pthread_attr_t *tattr,int scope);
```

Scope takes on the value of either `PTHREAD_SCOP_SYSTEM` or `PTHREAD_SCOPE_PROCESS`.

`pthread_attr_setscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred and an appropriate value is returned.

So to set a bound thread at thread creation on would do the following function calls:

```
#include <pthread.h>

pthread_attr_t attr;
pthread_t tid;
```

```
void start_routine;
void arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);
/* BOUND behavior */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

If the following conditions occur, the function fails and returns the corresponding value.

The function `pthread_attr_getscope()` is used to retrieve the thread scope, which indicates whether the thread is bound or unbound. It is prototyped by:

```
int pthread_attr_getscope(pthread_attr_t *tattr, int *scope);
```

An example use of this function is:

```
#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

If successful the approriate (`PTHREAD_SCOP_SYSTEM` or `PTHREAD_SCOPE_PROCESS`) wil be stored in `scope`.

`pthread_att_getscope()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

29.6 Thread Scheduling Policy

The POSIX draft standard specifies scheduling policy attributes of `SCHED_FIFO` (first-in-first-out), `SCHED_RR` (round-robin), or `SCHED_OTHER` (an implementation-defined method). `SCHED_FIFO` and `SCHED_RR` are optional in POSIX, and **only** are supported for *real time bound threads*.

Howver Note, currently, only the Solaris SCHED_OTHER default value is supported in pthreads. Attempting to set policy as SCHED_FIFO or SCHED_RR will result in the error ENOSUP.

The function is used to set the scheduling policy.It is prototyped by:

```
int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
```

pthread_attr_setschedpolicy() returns zero after completing successfully. Any other returned value indicates that an error occurred.

To set the scheduling policy to SCHED_OTHER simply do:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

There is a function pthread_attr_getschedpolicy() that retrieves the scheduling policy. But, currently, it is not of great use as it can only return the (Solaris-based) SCHED_OTHER default value

29.6.1 Thread Inherited Scheduling Policy

The function pthread_attr_setinheritsched() can be used to the inherited scheduling policy of a thread. It is prototyped by:

```
int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);
```

An inherit value of PTHREAD_INHERIT_SCHED (the default) means that the scheduling policies defined in the creating thread are to be used, and any scheduling attributes defined in the pthread_create() call are to be ignored. If PTHREAD_EXPLICIT_SCHED is used, the attributes from the pthread_create() call are to be used.

The function returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call of this function is:

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;

/* use the current scheduling policy */
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
```

The function `pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit)` may be used to inquire a current threads scheduling policy.

29.6.2 Set Scheduling Parameters

Scheduling parameters are defined in the `sched_param` structure; **only** priority `sched_param.sched_priority` is supported. This priority is an integer value the higher the value the higher a thread's proiority for scehduling. Newly created threads run with this priority. The `pthread_attr_setschedparam()` is used to set this stucture appropiately. It is prototyped by:

```
int pthread_attr_setschedparam(pthread_attr_t *tattr,
const struct sched_param *param);
```

and returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to `pthread_attr_setschedparam()` is:

```
#include <pthread.h>
pthread_attr_t tattr;
int newprio;
sched_param param;


/* set the priority; others are unchanged */
newprio = 30;
param.sched_priority = newprio;

/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);
```

The function `pthread_attr_getschedparam(pthread_attr_t *tattr, const struct sched_param *param)` may be used to inquire a current thread's priority of scheduling.

29.7 Thread Stack Size

Typically, thread stacks begin on page boundaries and any specified size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows result in sending a `SIGSEGV` signal to the offending thread. Thread stacks allocated by the caller are used as is.

When a stack is specified, the thread should also be created `PTHREAD_CREATE_JOINABLE`. That stack cannot be freed until the `pthread_join()` call for that thread has returned, because the thread's stack cannot be freed until the thread has terminated. The only reliable way to know if such a thread has terminated is through `pthread_join()`.

Generally, you do not need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of mmap to make the allocations.)

Each thread stack created by the threads library has a red zone. The library creates the red zone by appending a page to the top of a stack to catch stack overflows. This page is invalid and causes a memory fault if it is accessed. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

Note: Because runtime stack requirements vary, you should be absolutely certain that the specified stack will satisfy the runtime requirements needed for library calls and dynamic linking.

There are very few occasions when it is appropriate to specify a stack, its size, or both. It is difficult even for an expert to know if the right size was specified. This is because even a program compliant with ABI standards cannot determine its stack size statically. Its size is dependent on the needs of the particular runtime environment in which it executes.

29.7.1 Building Your Own Thread Stack

When you specify the size of a thread stack, be sure to account for the allocations needed by the invoked function and by each function called. The accounting should include calling sequence needs, local variables, and information structures.

Occasionally you want a stack that is a bit different from the default stack. An obvious situation is when the thread needs more than one megabyte of stack space. A less obvious situation is when the default stack is too large. You might be creating thousands of threads and not have enough virtual memory to handle the gigabytes of stack space that this many default stacks require.

The limits on the maximum size of a stack are often obvious, but what about the limits on its minimum size? There must be enough stack space to handle all of the stack frames that are pushed onto the stack, along with their local variables, and so on.

You can get the absolute minimum limit on stack size by calling the macro `PTHREAD_STACK_MIN` (defined in <pthread.h>), which returns the amount of stack space required for a thread that executes a `NULL` procedure. Useful threads need more than this, so be very careful when reducing the stack size.

The function `pthread_attr_setstacksize()` is used to set this a thread's stack size, it is prototyped by:

```
int pthread_attr_setstacksize(pthread_attr_t *tattr, int stacksize);
```

The `stacksize` attribute defines the size of the stack (in bytes) that the system will allocate. The size should not be less than the system-defined minimum stack size.

`pthread_attr_setstacksize()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

An example call to set the stacksize is:

```
#include <pthread.h>

pthread_attr_t tattr;
int stacksize;
int ret;
```

```
/* setting a new size */
stacksize = (PTHREAD_STACK_MIN + 0x4000);
ret = pthread_attr_setstacksize(&tattr, stacksize);
```

In the example above, size contains the size, in number of bytes, for the stack that the new thread uses. If size is zero, a default size is used. In most cases, a zero value works best. `PTHREAD_STACK_MIN` is the amount of stack space required to start a thread. This does not take into consideration the threads routine requirements that are needed to execute application code.

The function `pthread_attr_getstacksize(pthread_attr_t *tattr, size_t *size)` may be used to inquire about a current threads stack size as follows:

```
#include <pthread.h>

pthread_attr_t tattr;
int stacksize;
int ret;
/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &stacksize);
```

The current size of the stack is returned to the variable `stacksize`.

You may wish tp specify the base adress of thread's stack. The function `pthread_attr_setstackaddr()` does this task. It is prototyped by:

```
int pthread_attr_setstackaddr(pthread_attr_t *tattr,void *stackaddr);
```

The `stackaddr` parameter defines the base of the thread's stack. If this is set to non-null (NULL is the default) the system initializes the stack at that address.

The function returns zero after completing successfully. Any other returned value indicates that an error occurred.

This example shows how to create a thread with both a custom stack address and a custom stack size.

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
```

```
int ret;
void *stackbase;
int size = PTHREAD_STACK_MIN + 0x4000;
stackbase = (void *) malloc(size);
/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);
/* setting the base address in the attribute */
ret = pthread_attr_setstackaddr(&tattr, stackbase);
/* address and size specified */
ret = pthread_create(&tid, &tattr, func, arg);
```

The function `pthread_attr_getstackaddr(pthread_attr_t *tattr,void * *stackaddr)` can be used to obtain the base address for a current thread's stack address.

Chapter 30

Further Threads Programming:Synchronization

When we multiple threads running they will invariably need to communicate with each other in order *synchronise* their execution. This chapter describes the synchronization types available with threads and discusses when and how to use synchronization.

There are a few possible methods of synchronising threads:

- Mutual Exclusion (Mutex) Locks
- Condition Variables
- Semaphores

We will frequently make use of *Synchronization objects*: these are variables in memory that you access just like data. Threads in different processes can communicate with each other through synchronization objects placed in threads-controlled shared memory, even though the threads in different processes are generally invisible to each other.

Synchronization objects can also be placed in files and can have lifetimes beyond that of the creating process.

Here are some example situations that require or can profit from the use of synchronization:

- When synchronization is the only way to ensure consistency of shared data.

- When threads in two or more processes can use a single synchronization object jointly. Note that the synchronization object should be initialized by only one of the cooperating processes, because reinitializing a synchronization object sets it to the unlocked state.
- When synchronization can ensure the safety of mutable data.
- When a process can map a file and have a thread in this process get a record's lock. Once the lock is acquired, any other thread in any process mapping the file that tries to acquire the lock is blocked until the lock is released.
- Even when accessing a single primitive variable, such as an integer. On machines where the integer is not aligned to the bus data width or is larger than the data width, a single memory load can use more than one memory cycle. While this cannot happen on the SPARC architectures, portable programs cannot rely on this.

30.1 Mutual Exclusion Locks

Mutual exclusion locks (mutexes) are a comon method of serializing thread execution. Mutual exclusion locks synchronize threads, usually by ensuring that only one thread at a time executes a critical section of code. Mutex locks can also preserve single-threaded code.

Mutex attributes may be associated with every thread. To change the default mutex attributes, you can declare and initialize an mutex attribute object and then alter specific values much like we have seen in the last chapter on more general POSIX attributes. Often, the mutex attributes are set in one place at the beginning of the application so they can be located quickly and modified easily.

After the attributes for a mutex are configured, you initialize the mutex itself. Functions are available to initialize or destroy, lock or unlock, or try to lock a mutex.

30.1.1 Initializing a Mutex Attribute Object

The function `pthread_mutexattr_init()` is used to initialize attributes associated with this object to their default values. It is prototyped by:

```
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

Storage for each attribute object is allocated by the threads system during execution. `mattr` is an opaque type that contains a system-allocated attribute object. The possible values of mattr's scope are `PTHREAD_PROCESS_PRIVATE` (the default) and `PTHREAD_PROCESS_SHARED`.The default value of the pshared attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`, which means that the initialized mutex can be used within a process.

Before a mutex attribute object can be reinitialized, it must first be destroyed by `pthread_mutexattr_destroy()` (see below). The `pthread_mutexattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result. `pthread_mutexattr_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example of this function call is:

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

/* initialize an attribute to default value */

ret = pthread_mutexattr_init(&mattr);
```

30.1.2 Destroying a Mutex Attribute Object

The function `pthread_mutexattr_destroy()` deallocates the storage space used to maintain the attribute object created by `pthread_mutexattr_init()`. It is prototyped by:

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr);
```

which returns zero after completing successfully. Any other returned value indicates that an error occurred.

The function is called as follows:

```
#include <pthread.h>
```

```
pthread_mutexattr_t mattr;
int ret;

/* destroy an attribute */
ret = pthread_mutexattr_destroy(&mattr);
```

30.1.3 The Scope of a Mutex

The scope of a mutex variable can be either process private (intraprocess) or system wide (interprocess). The function `pthread_mutexattr_setpshared()` is used to set the scope of a mutex atrribute and it is prototype as follows:

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *mattr, int pshared);
```

If the mutex is created with the `pshared` (POSIX) attribute set to the `PTHREAD_PROCESS_SHARED` state, and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the `USYNC_PROCESS` flag in `mutex_init()` in Solaris threads. If the mutex `pshared` attribute is set to `PTHREAD_PROCESS_PRIVATE`, only those threads created by the same process can operate on the mutex. This is equivalent to the `USYNC_THREAD` flag in `mutex_init()` in Solaris threads.

`pthread_mutexattr_setpshared()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call is:

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

ret = pthread_mutexattr_init(&mattr);

/* resetting to its default value: private */
ret = pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_PRIVATE);
```

The function `pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr, int *pshared)` may be used to obtain the scope of a current thread mutex as follows:

```
 #include <pthread.h>
pthread_mutexattr_t mattr;
int pshared, ret;

/* get pshared of mutex */ ret =
pthread_mutexattr_getpshared(&mattr, &pshared);
```

30.1.4 Initializing a Mutex

The function `pthread_mutex_init()` to initialize the mutex, it is prototyped by:

```
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);
```

Here, `pthread_mutex_init()` initializes the mutex pointed at by `mp` to its default value if `mattr` is `NULL`, or to specify mutex attributes that have already been set with `pthread_mutexattr_init()`.

A mutex lock must not be reinitialized or destroyed while other threads might be using it. Program failure will result if either action is not done correctly. If a mutex is reinitialized or destroyed, the application must be sure the mutex is not currently in use. `pthread_mutex_init()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call is:

```
#include <pthread.h>

pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;

/* initialize a mutex to its default value */
ret = pthread_mutex_init(&mp, NULL);
```

When the mutex is initialized, it is in an unlocked state. The effect of `mattr` being `NULL` is the same as passing the address of a default mutex attribute object, but without the memory overhead. Statically defined mutexes can be initialized directly to have default attributes with the macro PTHREAD_MUTEX_INITIALIZER.

To initialise a mutex with non-default values do something like:

```
/* initialize a mutex attribute */
ret = pthread_mutexattr_init(&mattr);

/* change mattr default values with some function */
 ret = pthread_mutexattr_*();

/* initialize a mutex to a non-default value */
ret = pthread_mutex_init(&mp, &mattr);
```

30.1.5 Locking a Mutex

The function `pthread_mute_lock()` is used to lock a mutex, it is prototyped by:

```
int pthread_mutex_lock(pthread_mutex_t *mp);
```

`pthread_mute_lock()` locks the mutex pointed to by `mp`. When the mutex is already locked, the calling thread blocks and the mutex waits on a prioritized queue. When `pthread_mute_lock()` returns, the mutex is locked and the calling thread is the owner. `pthread_mute_lock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Therefor to lock a mutex `mp` on would do the following:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;

ret = pthread_mutex_lock(&mp);
```

To unlock a mutex use the function `pthread_mutex_unlock()` whose prototype is:

```
int pthread_mutex_unlock(pthread_mutex_t *mp);
```

Clearly, this function unlocks the mutex pointed to by `mp`.

The mutex must be locked and the calling thread **must** be the one that last locked the mutex (*i.e. the owner*). When any other threads are waiting for the mutex to become available, the thread at the head of the queue is unblocked. `pthread_mutex_unlock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A simple example call of `pthread_mutex_unlock()` is:

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

/* release the mutex */
ret = pthread_mutex_unlock(&mp);
```

Lock with a Nonblocking Mutex

The function `pthread_mutex_trylock()` to attempt to lock the mutex and is prototyped by:

```
int pthread_mutex_trylock(pthread_mutex_t *mp);
```

This function attempts to lock the mutex pointed to by `mp`. `pthread_mutex_trylock()` is a nonblocking version of `pthread_mutex_lock()`. When the mutex is already locked, this call returns with an error. Otherwise, the mutex is locked and the calling thread is the owner. `pthread_mutex_trylock()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

The function is called as follows:

```
#include <pthread.h>
pthread_mutex_t mp;

/* try to lock the mutex */
int ret; ret = pthread_ mutex_trylock(&mp);
```

30.1.6 Destroying a Mutex

The function `pthread_mutex_destroy()` may be used to destroy any state associated with the mutex. It is prototyped by:

```
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

and destroys a mutex pointed to by mp.

Note: that the space for storing the mutex is not freed. `pthread_mutex_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

It is called by:

```
#include <pthread.h>
pthread_mutex_t mp;
int ret;

/* destroy mutex */
ret = pthread_mutex_destroy(&mp);
```

30.1.7 Mutex Lock Code Examples

Here are some code fragments showing mutex locking.

Mutex Lock Example

We develop two small functions that use the mutex lock for different purposes.

- The `increment_count function()` uses the mutex lock simply to ensure an atomic update of the shared variable, `count`.
- The `get_count()` function uses the mutex lock to guarantee that the (`long long`) 64-bit quantity count is read atomically. On a 32-bit architecture, a long long is really two 32-bit quantities.

The 2 functions are as follows:

```
#include <pthread.h>
pthread_mutex_t count_mutex;
long long count;
```

```
void increment_count()
   { pthread\_mutex\_lock(&count_mutex);
     count = count + 1;
     pthread_mutex_unlock(&count_mutex);
   }


long long get_count()
   { long long c;
     pthread\_mutex\_lock(&count_mutex);
     c = count;
     pthread_mutex_unlock(&count_mutex);
     return (c);
    }
```

Recall that reading an integer value is an atomic operation because integer is the common word size on most machines.

Using Locking Hierarchies: Avoiding Deadlock

You may occasionally want to access two resources at once. For instance, you are using one of the resources, and then discover that the other resource is needed as well. However, there could be a problem if two threads attempt to claim both resources but lock the associated mutexes in different orders.

In this example, if the two threads lock mutexes 1 and 2 respectively, then a *deadlock* occurs when each attempts to lock the other mutex.

Thread 1	**Thread 2**

`/* use resource 1 */`
`pthread_mutex_lock(&m1);`
 	
`/* NOW use resources 2 + 1 */`
 	
`pthread_mutex_lock(&m2);`
 	
`pthread_mutex_lock(&m1);`
	`/* use resource 2 */`
	`pthread_mutex_lock(&m2);`
	
	`/* NOW use resources 1 + 2 */`
	
	`pthread_mutex_lock(&m1);`
	
	`pthread_mutex_lock(&m2);`

The best way to avoid this problem is to make sure that whenever threads lock multiple mutexes, they do so in the same order. This technique is known as lock hierarchies: order the mutexes by logically assigning numbers to them. Also, honor the restriction that you cannot take a mutex that is assigned n when you are holding any mutex assigned a number greater than n.

Note: The `lock_lint` tool can detect the sort of deadlock problem shown in this example.

The best way to avoid such deadlock problems is to use lock hierarchies. When locks are always taken in a prescribed order, deadlock should not occur. However, this technique cannot always be used :

- sometimes you must take the mutexes in an order other than prescribed.
- To prevent deadlock in such a situation, use `pthread_mutex_trylock()`. One thread must release its mutexes when it discovers that deadlock would otherwise be inevitable.

The idea of *Conditional Locking* use this approach:

Thread 1:

```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m2);

/* no processing */
pthread_mutex_unlock(&m2);
pthread_mutex_unlock(&m1);
```

Thread 2:

```
for (; ;) {
  pthread_mutex_lock(&m2);
  if(pthread_mutex_trylock(&m1)==0)
    /* got it! */
    break;
 /* didn't get it */
 pthread_mutex_unlock(&m2);
 }
/* get locks; no processing */
pthread_mutex_unlock(&m1);
pthread_mutex_unlock(&m2);
```

In the above example, thread 1 locks mutexes in the prescribed order, but thread 2 takes them out of order. To make certain that there is no deadlock, thread 2 has to take mutex 1 very carefully; if it were to block waiting for the mutex to be released, it is likely to have just entered into a deadlock with thread 1. To ensure this does not happen, thread 2 calls `pthread_mutex_trylock()`, which takes the mutex if it is available. If it is not, thread 2 returns immediately, reporting failure. At this point, thread 2 must release mutex 2, so that thread 1 can lock it, and then release both mutex 1 and mutex 2.

30.1.8 Nested Locking with a Singly Linked List

We have met basic linked structues in Section 10.3, when using threads which share a linked list structure the possibility of deadlock may arise.

By nesting mutex locks into the linked data structure and a simple ammendment of the link list code we can prevent deadlock by taking the locks in a prescribed order.

The modified linked is as follows:

```
typedef struct node1 {
     int value;
     struct node1 *link;
     pthread_mutex_t lock;
   } node1_t;
```

Note: we simply ammend a standard singly-linked list structure so that each node containing a mutex.

Assuming we have created a variable `node1_t ListHead`.

To remove a node from the list:

- first search the list starting at ListHead (which itself is never removed) until the desired node is found.
- To protect this search from the effects of concurrent deletions, lock each node before any of its contents are accessed.

  Because all searches start at ListHead, there is never a deadlock because the locks are always taken in list order.

- When the desired node is found, lock both the node and its predecessor since the change involves both nodes.

  Because the predecessor's lock is always taken first, you are again protected from deadlock.

The C code to remove an item from a singly linked list with nested locking is as follows:

```
node1_t *delete(int value)
  { node1_t *prev,
    *current; prev = &ListHead;

   pthread_mutex_lock(&prev->lock);
   while ((current = prev->link) != NULL)
     { pthread_mutex_lock(&current->lock);
       if (current->value == value)
         { prev->link = current->link;
           pthread_mutex_unlock(&current->lock);
           pthread_mutex_unlock(&prev->lock);
           current->link = NULL; return(current);
         }
       pthread_mutex_unlock(&prev->lock);
       prev = current;
      }
    pthread_mutex_unlock(&prev->lock);
    return(NULL);
   }
```

30.1.9 Solaris Mutex Locks

Similar mutual exclusion locks exist for in Solaris.

You should include the <synch.h> or <thread.h>libraries.

To initialize a mutex use `int mutex_init(mutex_t *mp, int type, void *arg))`. `mutex_init()` initializes the mutex pointed to by `mp`. The `type` can be one of the following (note that `arg` is currently ignored).

`USYNC_PROCESS` — The mutex can be used to synchronize threads in this and other processes.

`USYNC_THREAD` — The mutex can be used to synchronize threads in this process, only.

Mutexes can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be reinitialized while other threads might be using it.

The function `int mutex_destroy (mutex_t *mp)` destroys any state associated with the mutex pointed to by `mp`. **Note** that the space for storing the mutex is not freed.

To acquire a mutex lock use the function `mutex_lock(mutex_t *mp)` which locks the mutex pointed to by mp. When the mutex is already locked, the calling thread blocks until the mutex becomes available (blocked threads wait on a prioritized queue).

To release a mutex use `mutex_unlock(mutex_t *mp)` which unlocks the mutex pointed to by mp. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner).

To try to acquire a mutex use mutex_trylock(mutex_t *mp) to attempt to lock the mutex pointed to by `mp`. This function is a nonblocking version of `mutex_lock()`

30.2 Condition Variable Attributes

Condition variables can be usedto atomically block threads until a particular condition is true. Condition variables are *always* used in conjunction with mutex locks:

- With a condition variable, a thread can atomically block until a condition is satisfied.
- The condition is tested under the protection of a mutual exclusion lock (mutex).
  - When the condition is false, a thread usually blocks on a condition variable and atomically releases the mutex waiting for the condition to change.

  – When another thread changes the condition, it can signal the associated condition variable to cause one or more waiting threads to wake up, acquire the mutex again, and reevaluate the condition.

Condition variables can be used to synchronize threads among processes when they are allocated in memory that can be written to and is shared by the cooperating processes.

The scheduling policy determines how blocking threads are awakened. For the default SCHED_OTHER, threads are awakened in priority order. The attributes for condition variables must be set and initialized before the condition variables can be used.

As with mutex locks, The condiotion variable attributes must be initialised and set (or set to NULL) before an actual condition variable may be initialise (with appropriat attributes) and then used.

30.2.1 Initializing a Condition Variable Attribute

The function pthread_condattr_init() initializes attributes associated with this object to their default values. It is prototyped by:

```
int pthread_condattr_init(pthread_condattr_t *cattr);
```

Storage for each attribute object, cattr, is allocated by the threads system during execution. cattr is an opaque data type that contains a system-allocated attribute object. The possible values of cattr's scope are PTHREAD_PROCESS_PRIVATE and PTHREAD_PROCESS_SHARED. The default value of the pshared attribute when this function is called is PTHREAD_PROCESS_PRIVATE, which means that the initialized condition variable can be used within a process.

Before a condition variable attribute can be reused, it must first be reinitialized by pthread_condattr_destroy(). The pthread_condattr_init() call returns a pointer to an opaque object. If the object is not destroyed, a memory leak will result.

pthread_condattr_init() returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

A simple example call of this function is :

```
#include <pthread.h>

pthread_condattr_t cattr;
int ret;

/* initialize an attribute to default value */
ret = pthread_condattr_init(&cattr);
```

30.2.2 Destoying a Condition Variable Attribute

The function `pthread_condattr_destroy()` removes storage and renders the attribute object invalid, it is prototyped by:

```
int pthread_condattr_destroy(pthread_condattr_t *cattr);
```

`pthread_condattr_destroy()` returns zero after completing successfully and destroying the condition variable pointed to by `cattr`. Any other returned value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

30.2.3 The Scope of a Condition Variable

The scope of a condition variable can be either process private (intraprocess) or system wide (interprocess), as with mutex locks. If the condition variable is created with the pshared attribute set to the `PTHREAD_PROCESS_SHARED` state, and it exists in shared memory, it can be shared among threads from more than one process. This is equivalent to the `USYNC_PROCESS` flag in `mutex_init()` in the original Solaris threads. If the mutex pshared attribute is set to `PTHREAD_PROCESS_PRIVATE` (default value), only those threads created by the same process can operate on the mutex. Using `PTHREAD_PROCESS_PRIVATE` results in the same behavior as with the `USYNC_THREAD` flag in the original Solaris threads `cond_init()` call, which is that of a local condition variable. `PTHREAD_PROCESS_SHARED` is equivalent to a global condition variable.

The function `pthread_condattr_setpshared()` is used to set the scope of a condition variable, it is prototyped by:

```
int pthread_condattr_setpshared(pthread_condattr_t *cattr, int pshared);
```

The condition variable attribute `cattr` must be initialised first and the value of `pshared` is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

`pthread_condattr_setpshared()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

A sample use of this function is as follows:

```
#include <pthread.h>

pthread_condattr_t cattr;
int ret;

/* Scope: all processes */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

/* OR */
/* Scope: within a process */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

The function `int pthread_condattr_getpshared(const pthread_condattr_t *cattr, int *pshared)` may be used to obtain the scope of a given condition variable.

30.2.4 Initializing a Condition Variable

The function `pthread_cond_init()` initializes the condition variable and is prototyped as follows:

```
int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);
```

The condition variable which is initialized is pointed at by `cv` and is set to its default value if `cattr` is `NULL`, or to specific `cattr` condition variable attributes that are already set with `pthread_condattr_init()`. The effect of `cattr` being NULL is the same as passing the address of a default condition variable attribute object, but without the memory overhead.

Statically-defined condition variables can be initialized directly to have default attributes with the macro `PTHREAD_COND_INITIALIZER`. This has the same effect as dynamically allocating `pthread_cond_init()` with null attributes. No error checking is done. Multiple threads must not simultaneously initialize or reinitialize the same condition variable. If a condition variable is reinitialized or destroyed, the application must be sure the condition variable is not in use.

pthread_cond_init() returns zero after completing successfully. Any other returned value indicates that an error occurred.

Sample calls of this function are:

```
#include <pthread.h>

pthread_cond_t cv;
pthread_condattr_t cattr;
int ret;

/* initialize a condition variable to its default value */
ret = pthread_cond_init(&cv, NULL);

/* initialize a condition variable */ ret =
pthread_cond_init(&cv, &cattr);
```

30.2.5 Block on a Condition Variable

The function pthread_cond_wait() is used to atomically release a mutex and to cause the calling thread to block on the condition variable. It is protoyped by:

```
int pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);
```

The mutex that is released is pointed to by mutex and the condition variable pointed to by cv is blocked.

pthread_cond_wait() returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

A simple example call is:

```
#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mutex;
```

```
int ret;

/* wait on condition variable */
ret = pthread_cond_wait(&cv, &mutex);
```

The blocked thread can be awakened by a `pthread_cond_signal()`, a `pthread_cond_broadcast()`, or when interrupted by delivery of a signal. Any change in the value of a condition associated with the condition variable cannot be inferred by the return of `pthread_cond_wait()`, and any such condition must be reevaluated. The `pthread_cond_wait()` routine always returns with the mutex locked and owned by the calling thread, even when returning an error. This function blocks until the condition is signaled. It atomically releases the associated mutex lock before blocking, and atomically acquires it again before returning. In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when it changes the condition value. This causes one or all of the threads waiting on the condition to unblock and to try to acquire the mutex lock again. Because the condition can change before an awakened thread returns from `pthread_cond_wait()`, the condition that caused the wait must be retested before the mutex lock is acquired.

The recommended test method is to write the condition check as a while loop that calls `pthread_cond_wait()`, as follows:

```
pthread_mutex_lock();

while(condition_is_false)
  pthread_cond_wait();
pthread_mutex_unlock();
```

No specific order of acquisition is guaranteed when more than one thread blocks on the condition variable. Note also that `pthread_cond_wait()` is a cancellation point. If a cancel is pending and the calling thread has cancellation enabled, the thread terminates and begins executing its cleanup handlers while continuing to hold the lock.

To unblock a specific thread use `pthread_cond_signal()` which is prototyped by:

```
int pthread_cond_signal(pthread_cond_t *cv);
```

This unblocks one thread that is blocked on the condition variable pointed to by `cv`. `pthread_cond_signal()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

You should always call `pthread_cond_signal()` under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait. The scheduling policy determines the order in which blocked threads are awakened. For `SCHED_OTHER`, threads are awakened in priority order. When no threads are blocked on the condition variable, then calling `pthread_cond_signal()`l has no effect.

The folloowing code fragment illustrates how to avoid an infinite problem described above:

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count()
   { pthread_mutex_lock(&count_lock);

     while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
     count = count - 1;
     pthread_mutex_unlock(&count_lock);
   }

increment_count()
   { pthread_mutex_lock(&count_lock);
     if (count == 0)
       pthread_cond_signal(&count_nonzero);
     count = count + 1;
     pthread_mutex_unlock(&count_lock);
    }
```

You can also block until a specified event occurs. The function `pthread_cond_timedwait()` is used for this purpose. It is prototyped by:

```
int pthread_cond_timedwait(pthread_cond_t *cv,
      pthread_mutex_t *mp, const struct timespec *abstime);
```

`pthread_cond_timedwait()` is used in a similar manner to `pthread_cond_wait()`: `pthread_cond_timedwait()` blocks until the condition is signaled or until the time of day, specified by `abstime`, has passed. `pthread_cond_timedwait()` always returns with the mutex, `mp`, locked and owned by the calling thread, even when it is returning an error. `pthread_cond_timedwait()` is also a cancellation point.

`pthread_cond_timedwait()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

An examle call of this function is:

```
#include <pthread.h>
#include <time.h>

pthread_timestruc_t to;
pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;

/* wait on condition variable */

ret = pthread_cond_timedwait(&cv, &mp, &abstime);


pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;

while (cond == FALSE)
  {  err = pthread_cond_timedwait(&c, &m, &to);
     if (err == ETIMEDOUT)
       { /* timeout, do something */
         break;
       }
    }
pthread_mutex_unlock(&m);
```

All threads may be unblocked in one function: `pthread_cond_broadcast()`. This function is prototyped as follows:

```
int pthread_cond_broadcast(pthread_cond_t *cv);
```

`pthread_cond_broadcast()` unblocks all threads that are blocked on the condition variable pointed to by `cv`, specified by `pthread_cond_wait()`. When no threads are blocked on the condition variable, `pthread_cond_broadcast()` has no effect.

`pthread_cond_broadcast()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

Since `pthread_cond_broadcast()` causes all threads blocked on the condition to contend again for the mutex lock, use carefully. For example, use `pthread_cond_broadcast()` to allow threads to contend for varying resource amounts when resources are freed:

```
#include <pthread.h>

pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
  { pthread_mutex_lock(&rsrc_lock);
    while (resources < amount)
       pthread_cond_wait(&rsrc_add, &rsrc_lock);

   resources -= amount;
   pthread_mutex_unlock(&rsrc_lock);
  }

add_resources(int amount)
  { pthread_mutex_lock(&rsrc_lock);
```

```
    resources += amount;
    pthread_cond_broadcast(&rsrc_add);
    pthread_mutex_unlock(&rsrc_lock);
  }
```

Note: that in `add_resources` it does not matter whether resources is updated first or if `pthread_cond_broadcast()` is called first inside the mutex lock. Call `pthread_cond_broadcast()` under the protection of the same mutex that is used with the condition variable being signaled. Otherwise, the condition variable could be signaled between the test of the associated condition and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

30.2.6 Destroying a Condition Variable State

The function `pthread_cond_destroy()` to destroy any state associated with the condition variable, it is prototyped by:

```
int pthread_cond_destroy(pthread_cond_t *cv);
```

The condition variable pointed to by `cv` will be destroyed by this call:

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* Condition variable is destroyed */
ret = pthread_cond_destroy(&cv);
```

Note that the space for storing the condition variable is not freed.

`pthread_cond_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

30.2.7 Solaris Condition Variables

Similar condition variables exist in Solaris. The functions are prototyped in <thread.h>.

To initialize a condition variable use `int cond_init(cond_t *cv, int type, int arg)` which initializes the condition variable pointed to by `cv`. The `type` can be one of `USYNC_PROCESS` or `USYNC_THREAD` (See Solaris mutex (Section 30.1.9 for more details). Note that `arg` is currently ignored.

Condition variables can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed. Multiple threads must not initialize the same condition variable simultaneously. A condition variable must not be reinitialized while other threads might be using it.

To destroy a condition variable use `int cond_destroy(cond_t *cv)` which destroys a state associated with the condition variable pointed to by `cv`. The space for storing the condition variable is not freed.

To wait for a condition use `int cond_wait(cond_t *cv, mutex_t *mp)` which atomically releases the mutex pointed to by `mp` and to cause the calling thread to block on the condition variable pointed to by `cv`.

The blocked thread can be awakened by `cond_signal(cond_t *cv)`, `cond_broadcast(cond_t *cv)`, or when interrupted by delivery of a signal or a fork. Use `cond_signal()` to unblock one thread that is blocked on the condition variable pointed to by `cv`. Call this function under protection of the same mutex used with the condition variable being signaled. Otherwise, the condition could be signaled between its test and `cond_wait()`, causing an infinite wait. Use `cond_broadcast()` to unblock all threads that are blocked on the condition variable pointed to by `cv`. When no threads are blocked on the condition variable then `cond_broadcast()` has no effect.

Finally, to wait until the condition is signaled or for an absolute time use `int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime)` Use `cond_timedwait()` as you would use `cond_wait()`, except that `cond_timedwait()` does not block past the time of day specified by abstime. `cond_timedwait()` always returns with the mutex locked and owned by the calling thread even when returning an error.

30.3 Threads and Semaphores

30.3.1 POSIX Semaphores

Chapter 25 has dealt with semaphore programming for POSIX and System V IPC semaphores.

Semaphore operations are the same in both POSIX and Solaris. The function names are changed from `sema_` in Solaris to `sem_` in pthreads. Solaris semaphore are defined in <thread.h>.

In this section we give a brief description of Solaris thread semaphores.

30.3.2 Basic Solaris Semaphore Functions

To initialize the function `int sema_init(sema_t *sp, unsigned int count, int type, void *arg)` is used. `sema`. `type` can be one of the following):

`USYNC_PROCESS` — The semaphore can be used to synchronize threads in this process and other processes. Only one process should initialize the semaphore.

`USYNC_THREAD` — The semaphore can be used to synchronize threads in this process.

`arg` is currently unused.

Multiple threads **must not** initialize the same semaphore simultaneously. A semaphore **must not** be reinitialized while other threads may be using it.

To increment a Semaphore use the function `int sema_post(sema_t *sp)`. `sema_post` atomically increments the semaphore pointed to by `sp`. When any threads are blocked on the semaphore, one is unblocked.

To block on a Semaphore use `int sema_wait(sema_t *sp)`. `sema_wait()` to block the calling thread until the count in the semaphore pointed to by `sp` becomes greater than zero, then atomically decrement it.

To decrement a Semaphore count use `int sema_trywait(sema_t *sp)`. `sema_trywait()` atomically decrements the count in the semaphore pointed to by `sp` when the count is greater than zero. This function is a nonblocking version of `sema_wait()`.

To destroy the Semaphore state call the function `sema_destroy(sema_t *sp)`. `sema_destroy()` to destroy any state associated with the semaphore pointed to by `sp`. The space for storing the semaphore is not freed.

Chapter 31

Thread programming examples

This chapter gives some full code examples of thread programs. These examles are taken from a variety of sources:

- The sun workshop developers web page *http://www.sun.com/workshop/threads/share-code/* on threads is an excelleny source
- The web page *http://www.sun.com/workshop/threads/Berg-Lewis/examples.html* where example from the *Threads Primer* Book by D. Berg anD B. Lewis are also a major resource.

31.1 Using `thr_create()` and `thr_join()`

This example exercises the `thr_create()` and `thr_join()` calls. There is not a parent/child relationship between threads as there is for processes. This can easily be seen in this example, because threads are created and joined by many different threads in the process. The example also shows how threads behave when created with different attributes and options.

Threads can be created by any thread and joined by any other.

The main thread: In this example the main thread's sole purpose is to create new threads. Threads A, B, and C are created by the main thread. Notice that thread B is created suspended. After creating the new threads, the main thread exits. Also notice that the main thread exited by calling t̂hr_exit(). If the main thread had used the exit() call, the whole process would have exited. The main thread's exit status and resources are held until it is joined by thread C.

Thread A: The first thing thread A does after it is created is to create thread D. Thread A then simulates some processing and then exits, using `thr_exit()`. Notice that thread A was created with the `THR_DETACHED` flag, so thread A's resources will be immediately reclaimed upon its exit. There is no way for thread A's exit status to be collected by a `thr_join()` call.

Thread B: Thread B was created in a suspended state, so it is not able to run until thread D continues it by making the `thr_continue()` call. After thread B is continued, it simulates some processing and then exits. Thread B's exit status and thread resources are held until joined by thread E.

Thread C: The first thing that thread C does is to create thread F. Thread C then joins the main thread. This action will collect the main thread's exit status and allow the main thread's resources to be reused by another thread. Thread C will block, waiting for the main thread to exit, if the main thread has not yet called `thr_exit()`. After joining the main thread, thread C will simulate some processing and then exit. Again, the exit status and thread resources are held until joined by thread E.

Thread D: Thread D immediately creates thread E. After creating thread E, thread D continues thread B by making the `thr_continue()` call. This call will allow thread B to start its execution. Thread D then tries to join thread E, blocking until thread E has exited. Thread D then simulates some processing and exits. If all went well, thread D should be the last nondaemon thread running. When thread D exits, it should do two things: stop the execution of any daemon threads and stop the execution of the process.

Thread E: Thread E starts by joining two threads, threads B and C. Thread E will block, waiting for each of these thread to exit. Thread E will then simulate some processing and will exit. Thread E's exit status and thread resources are held by the operating system until joined by thread D.

Thread F: Thread F was created as a bound, daemon thread by using the `THR_BOUND` and `THR_DAEMON` flags in the `thr_create()` call. This means that it will run on its own LWP until all the nondaemon threads have exited the process. This type of thread can be used when you want some type of "background" processing to always be running, except when all the "regular" threads have exited the process. If thread F was created as a non-daemon thread, then it would continue to run forever, because a process will continue while there is at least one thread still running. Thread F will exit when all the nondaemon threads have exited. In this case, thread D should be the last nondaemon thread running, so when thread D exits, it will also cause thread F to exit.

This example, however trivial, shows how threads behave differently, based on their creation options. It also shows what happens on the exit of a thread, again based on how it was created. If you understand this example and how it flows, you should have a good understanding of how to use `thr_create()` and `thr_join()` in your own programs. Hopefully you can also see how easy it is to create and join threads.

The source to `multi_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* Function prototypes for thread routines */
void *sub_a(void *);
void *sub_b(void *);
void *sub_c(void *);
void *sub_d(void *);
void *sub_e(void *);
void *sub_f(void *);

thread_t thr_a, thr_b, thr_c;

void main()
{
thread_t main_thr;

main_thr = thr_self();
printf("Main thread = %d\n", main_thr);

if (thr_create(NULL, 0, sub_b, NULL, THR_SUSPENDED|THR_NEW_LWP, &thr_b))
        fprintf(stderr,"Can't create thr_b\n"), exit(1);

if (thr_create(NULL, 0, sub_a, (void *)thr_b, THR_NEW_LWP, &thr_a))
        fprintf(stderr,"Can't create thr_a\n"), exit(1);

if (thr_create(NULL, 0, sub_c, (void *)main_thr, THR_NEW_LWP, &thr_c))
        fprintf(stderr,"Can't create thr_c\n"), exit(1);
printf("Main Created threads A:%d B:%d C:%d\n", thr_a, thr_b, thr_c);
printf("Main Thread exiting...\n");
thr_exit((void *)main_thr);
}

void *sub_a(void *arg)
{
thread_t thr_b = (thread_t) arg;
thread_t thr_d;
int i;

printf("A: In thread A...\n");

if (thr_create(NULL, 0, sub_d, (void *)thr_b, THR_NEW_LWP, &thr_d))
        fprintf(stderr, "Can't create thr_d\n"), exit(1);

printf("A: Created thread D:%d\n", thr_d);

/* process
*/
for (i=0;i<1000000*(int)thr_self();i++);
printf("A: Thread exiting...\n");
thr_exit((void *)77);
}

void * sub_b(void *arg)
{
int i;

printf("B: In thread B...\n");

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("B: Thread exiting...\n");
thr_exit((void *)66);
}
```

```
void * sub_c(void *arg)
{
void *status;
int i;
thread_t main_thr, ret_thr;

main_thr = (thread_t)arg;

printf("C: In thread C...\n");

if (thr_create(NULL, 0, sub_f, (void *)0, THR_BOUND|THR_DAEMON, NULL))
        fprintf(stderr, "Can't create thr_f\n"), exit(1);

printf("C: Join main thread\n");

if (thr_join(main_thr,(thread_t *)&ret_thr, &status))
        fprintf(stderr, "thr_join Error\n"), exit(1);

printf("C: Main thread (%d) returned thread (%d) w/status %d\n", main_thr, ret_thr

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("C: Thread exiting...\n");
thr_exit((void *)88);
}


void * sub_d(void *arg)
{
thread_t thr_b = (thread_t) arg;
int i;
thread_t thr_e, ret_thr;
void *status;

printf("D: In thread D...\n");
```

```
if (thr_create(NULL, 0, sub_e, NULL, THR_NEW_LWP, &thr_e))
        fprintf(stderr,"Can't create thr_e\n"), exit(1);

printf("D: Created thread E:%d\n", thr_e);
printf("D: Continue B thread = %d\n", thr_b);

thr_continue(thr_b);
printf("D: Join E thread\n");

if(thr_join(thr_e,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("D: E thread (%d) returned thread (%d) w/status %d\n", thr_e,
ret_thr, (int) status);

/* process
*/

for (i=0;i<1000000*(int)thr_self();i++);
printf("D: Thread exiting...\n");
thr_exit((void *)55);
}


void * sub_e(void *arg)
{
int i;
thread_t ret_thr;
void *status;

printf("E: In thread E...\n");
printf("E: Join A thread\n");

if(thr_join(thr_a,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: A thread (%d) returned thread (%d) w/status %d\n", ret_thr, ret_
printf("E: Join B thread\n");

if(thr_join(thr_b,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: B thread (%d) returned thread (%d) w/status %d\n", thr_b, ret_thr, (int
printf("E: Join C thread\n");

if(thr_join(thr_c,(thread_t *)&ret_thr, &status))
        fprintf(stderr,"thr_join Error\n"), exit(1);

printf("E: C thread (%d) returned thread (%d) w/status %d\n", thr_c, ret_thr, (int

for (i=0;i<1000000*(int)thr_self();i++);

printf("E: Thread exiting...\n");
thr_exit((void *)44);
}


void *sub_f(void *arg)
{
int i;

printf("F: In thread F...\n");

while (1) {
        for (i=0;i<10000000;i++);
        printf("F: Thread F is still running...\n");
        }
}
```

31.2 Arrays

This example uses a data structure that contains multiple arrays of data. Multiple threads will concurrently vie for access to the arrays. To control this access, a mutex variable is used within the data structure to lock the entire array and serialize the access to the data.

The main thread first initializes the data structure and the mutex variable. It then sets a level of concurrency and creates the worker threads. The main thread then blocks by joining all the threads. When all the threads have exited, the main thread prints the results.

The worker threads modify the shared data structure from within a loop. Each time the threads need to modify the shared data, they lock the mutex variable associated with the shared data. After modifying the data, the threads unlock the mutex, allowing another thread access to the data.

This example may look quite simple, but it shows how important it is to control access to a simple, shared data structure. The results can be quite different if the mutex variable is not used.

The source to `array.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* sample array data structure */
struct {
        mutex_t data_lock[5];
        int     int_val[5];
        float   float_val[5];
        } Data;

/* thread function */
void *Add_to_Value();

main()
{
int i;

/* initialize the mutexes and data */
for (i=0; i<5; i++) {
        mutex_init(&Data.data_lock[i], USYNC_THREAD, 0);
        Data.int_val[i] = 0;
        Data.float_val[i] = 0;
        }
/* set concurrency and create the threads */
thr_setconcurrency(4);

for (i=0; i<5; i++)
    thr_create(NULL, 0, Add_to_Value, (void *)(2*i), 0, NULL);

/* wait till all threads have finished */
for (i=0; i<5; i++)
        thr_join(0,0,0);

/* print the results */
printf("Final Values.....\n");

for (i=0; i<5; i++) {
        printf("integer value[%d] =\t%d\n", i, Data.int_val[i]);
        printf("float value[%d] =\t%.0f\n\n", i, Data.float_val[i]);
        }

return(0);
}


/* Threaded routine */
void *Add_to_Value(void *arg)
{
int inval = (int) arg;
int i;

for (i=0;i<10000;i++){
    mutex_lock(&Data.data_lock[i%5]);
       Data.int_val[i%5] += inval;
       Data.float_val[i%5] += (float) 1.5 * inval;
    mutex_unlock(&Data.data_lock[i%5]);
    }

return((void *)0);
}
```

31.3 Deadlock

This example demonstrates how a deadlock can occur in multithreaded programs that use synchronization variables. In this example a thread is created that continually adds a value to a global variable. The thread uses a mutex lock to protect the global data.

The main thread creates the counter thread and then loops, waiting for user input. When the user presses the Return key, the main thread suspends the counter thread and then prints the value of the global variable. The main thread prints the value of the global variable under the protection of a mutex lock.

The problem arises in this example when the main thread suspends the counter thread while the counter thread is holding the mutex lock. After the main thread suspends the counter thread, it tries to lock the mutex variable. Since the mutex variable is already held by the counter thread, which is suspended, the main thread deadlocks.

This example may run fine for a while, as long as the counter thread just happens to be suspended when it is not holding the mutex lock. The example demonstrates how tricky some programming issues can be when you deal with threads.

The source to `susp_lock.c`

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>

/* Prototype for thread subroutine */
void *counter(void *);

int count;
mutex_t count_lock;

main()
{
char str[80];
thread_t ctid;

/* create the thread counter subroutine */
thr_create(NULL, 0, counter, 0, THR_NEW_LWP|THR_DETACHED, &ctid);

while(1) {
        gets(str);
        thr_suspend(ctid);

        mutex_lock(&count_lock);
        printf("\n\nCOUNT = %d\n\n", count);
        mutex_unlock(&count_lock);

        thr_continue(ctid);
        }

return(0);
}

void *counter(void *arg)
{
int i;

while (1) {
        printf("."); fflush(stdout);

        mutex_lock(&count_lock);
        count++;

        for (i=0;i<50000;i++);

        mutex_unlock(&count_lock);

        for (i=0;i<50000;i++);
        }

return((void *)0);
}
```

31.4 Signal Handler

This example shows how easy it is to handle signals in multithreaded programs. In most programs, a different signal handler would be needed to service each type of signal that you wanted to catch. Writing each of the signal handlers can be time consuming and can be a real pain to debug.

This example shows how you can implement a signal handler thread that will service all asynchronous signals that are sent to your process. This is an easy way to deal with signals, because only one thread is needed to handle all the signals. It also makes it easy when you create new threads within the process, because you need not worry about signals in any of the threads.

First, in the main thread, mask out all signals and then create a signal handling thread. Since threads inherit the signal mask from their creator, any new threads created after the new signal mask will also mask all signals. This idea is key, because the only thread that will receive signals is the one thread that does not block all the signals.

The signal handler thread waits for all incoming signals with the sigwait() call. This call unmasks the signals given to it and then blocks until a signal arrives. When a signal arrives, sigwait() masks the signals again and then returns with the signal ID of the incoming signal.

You can extend this example for use in your application code to handle all your signals. Notice also that this signal concept could be added in your existing nonthreaded code as a simpler way to deal with signals.

The source to `thr_sig.c`

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <signal.h>
#include <sys/types.h>

void *signal_hand(void *);

main()
{
sigset_t set;

/* block all signals in main thread.  Any other threads that are
```

```
   created after this will also block all signals */

sigfillset(&set);

thr_sigsetmask(SIG_SETMASK, &set, NULL);

/* create a signal handler thread.  This thread will catch all
   signals and decide what to do with them.  This will only
   catch nondirected signals.  (I.e., if a thread causes a SIGFPE
   then that thread will get that signal. */

thr_create(NULL, 0, signal_hand, 0, THR_NEW_LWP|THR_DAEMON|THR_DETACHED, NULL);

while (1) {
/*
Do your normal processing here....
*/
}  /* end of while */

return(0);
}

void *signal_hand(void *arg)
{
sigset_t set;
int sig;

sigfillset(&set); /* catch all signals */

while (1) {
  /* wait for a signal to arrive */

switch (sig=sigwait(&set)) {

  /* here you would add whatever signal you needed to catch */
  case SIGINT : {
printf("Interrupted with signal %d, exiting...\n", sig);
exit(0);
}

  default : printf("GOT A SIGNAL = %d\n", sig);
  } /* end of switch */
} /* end of while */

return((void *)0);
} /* end of signal_hand */
```

Another example of a signal handler, `sig_kill.c`:

```
/*
*  Multithreaded Demo Source
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
*  Users may copy, modify or distribute this file at will.
*
*  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
*  THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
*  PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
*
*  This file is provided with no support and without any obligation on the
*  part of SunSoft, Inc. to assist in its use, correction, modification or
*  enhancement.
*
*  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT
*  TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
*  FILE OR ANY PART THEREOF.
*
*  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY
*  LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL
*  DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH
*  DAMAGES.
*
*  SunSoft, Inc.
*  2550 Garcia Avenue
*  Mountain View, California  94043
*/

/*
 * Rich Schiavi writes:   Sept 11, 1994
 *
 * I believe the recommended way to kill certain threads is
 * using a signal handler which then will exit that particular
 * thread properly. I'm not sure the exact reason (I can't remember), but
 * if you take out the signal_handler routine in my example, you will see what
 * you describe, as the main process dies even if you send the
 * thr_kill to the specific thread.

 * I whipped up a real quick simple example which shows this using
 * some sleep()s to get a good simulation.
 */

#include <stdio.h>
#include <thread.h>
#include <signal.h>

static thread_t  one_tid, two_tid, main_thread;
static  void *first_thread();
static  void *second_thread();
void ExitHandler(int);

static mutex_t      first_mutex, second_mutex;
int  first_active = 1 ;
int second_active = 1;

main()

{
  int i;
  struct sigaction act;

  act.sa_handler = ExitHandler;
  (void) sigemptyset(&act.sa_mask);
  (void) sigaction(SIGTERM, &act, NULL);

  mutex_init(&first_mutex, 0 , 0);
  mutex_init(&second_mutex, 0 , 0);
  main_thread = thr_self();

  thr_create(NULL,0,first_thread,0,THR_NEW_LWP,&one_tid);
  thr_create(NULL,0,second_thread,0,THR_NEW_LWP,&two_tid);

  for (i = 0; i < 10; i++){
    fprintf(stderr, "main loop: %d\n", i);
    if (i == 5) {
      thr_kill(one_tid, SIGTERM);
    }
    sleep(3);
  }
  thr_kill(two_tid, SIGTERM);
  sleep(5);
  fprintf(stderr, "main exit\n");
}

static void *first_thread()
{
  int i = 0;

  fprintf(stderr, "first_thread id: %d\n", thr_self());
  while (first_active){
    fprintf(stderr, "first_thread: %d\n", i++);
    sleep(2);
  }
  fprintf(stderr, "first_thread exit\n");
}

static void *second_thread()
```

```
{
  int i = 0;

  fprintf(stderr, "second_thread id: %d\n", thr_self());

  while (second_active){
    fprintf(stderr, "second_thread: %d\n", i++);
    sleep(3);
  }
  fprintf(stderr, "second_thread exit\n");
}

void ExitHandler(int sig)
{
  thread_t id;

  id = thr_self();

  fprintf(stderr, "ExitHandler thread id: %d\n", id);
  thr_exit(0);

}
```

31.5 Interprocess Synchronization

This example uses some of the synchronization variables available in the threads library to synchronize access to a resource shared between two processes. The synchronization variables used in the threads library are an advantage over standard IPC synchronization mechanisms because of their speed. The synchronization variables in the threads libraries have been tuned to be very lightweight and very fast. This speed can be an advantage when your application is spending time synchronizing between processes.

This example shows how semaphores from the threads library can be used between processes. Note that this program does not use threads; it is just using the lightweight semaphores available from the threads library.

When using synchronization variables between processes, it is important to make sure that only one process initializes the variable. If both processes try to initialize the synchronization variable, then one of the processes will overwrite the state of the variable set by the other process.

The source to `ipc.c`

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <synch.h>
#include <sys/types.h>
#include <unistd.h>

/* a structure that will be used between processes */
typedef struct {
sema_t mysema;
int num;
} buf_t;

main()
{
int  i, j, fd;
buf_t  *buf;

/* open a file to use in a memory mapping */
fd = open("/dev/zero", O_RDWR);

/* create a shared memory map with the open file for the data
   structure that will be shared between processes */
buf=(buf_t *)mmap(NULL, sizeof(buf_t), PROT_READ|PROT_WRITE, MAP_SHARED, fd

/* initialize the semaphore -- note the USYNC_PROCESS flag; this makes
   the semaphore visible from a process level */
sema_init(&buf->mysema, 0, USYNC_PROCESS, 0);

/* fork a new process */
if (fork() == 0) {
/* The child will run this section of code */
for (j=0;j<5;j++)
{
/* have the child "wait" for the semaphore */

printf("Child PID(%d): waiting...\n", getpid());
sema_wait(&buf->mysema);

/* the child decremented the semaphore */

printf("Child PID(%d): decrement semaphore.\n", getpid());
}
/* exit the child process */
  printf("Child PID(%d): exiting...\n", getpid());
exit(0);
}

/* The parent will run this section of code */
/* give the child a chance to start running */

sleep(2);

for (i=0;i<5;i++)
{
/* increment (post) the semaphore */

printf("Parent PID(%d): posting semaphore.\n", getpid());
sema_post(&buf->mysema);

/* wait a second */
sleep(1);
}

/* exit the parent process */
printf("Parent PID(%d): exiting...\n", getpid());

return(0);
}
```

31.6 The Producer / Consumer Problem

This example will show how condition variables can be used to control access of reads and writes to a buffer. This example can also be thought as a producer/consumer problem, where the producer adds items to the buffer and the consumer removes items from the buffer.

Two condition variables control access to the buffer. One condition variable is used to tell if the buffer is full, and the other is used to tell if the buffer is empty. When the producer wants to add an item to the buffer, it checks to see if the buffer is full; if it is full the producer blocks on the `cond_wait()` call, waiting for an item to be removed from the buffer. When the consumer removes an item from the buffer, the buffer is no longer full, so the producer is awakened from the `cond_wait()` call. The producer is then allowed to add another item to the buffer.

The consumer works, in many ways, the same as the producer. The consumer uses the other condition variable to determine if the buffer is empty. When the consumer wants to remove an item from the buffer, it checks to see if it is empty. If the buffer is empty, the consumer then blocks on the `cond_wait()` call, waiting for an item to be added to the buffer. When the producer adds an item to the buffer, the consumer's condition is satisfied, so it can then remove an item from the buffer.

The example copies a file by reading data into a shared buffer (producer) and then writing data out to the new file (consumer). The Buf data structure is used to hold both the buffered data and the condition variables that control the flow of the data.

The main thread opens both files, initializes the `Buf` data structure, creates the consumer thread, and then assumes the role of the producer. The producer reads data from the input file, then places the data into an open buffer position. If no buffer positions are available, then the producer waits via the `cond_wait()` call. After the producer has read all the data from the input file, it closes the file and waits for (joins) the consumer thread.

The consumer thread reads from a shared buffer and then writes the data to the output file. If no buffers positions are available, then the consumer waits for the producer to fill a buffer position. After the consumer has read all the data, it closes the output file and exits.

If the input file and the output file were residing on different physical disks, then this example could execute the reads and writes in parallel. This parallelism would significantly increase the throughput of the example through the use of threads.

The source to `prod_cons.c`:

```
#define _REEENTRANT
#include <stdio.h>
#include <thread.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>

#define BUFSIZE 512
#define BUFCNT  4

/* this is the data structure that is used between the producer
   and consumer threads */

struct {
        char buffer[BUFCNT][BUFSIZE];
        int byteinbuf[BUFCNT];
        mutex_t buflock;
        mutex_t donelock;
        cond_t adddata;
        cond_t remdata;
        int nextadd, nextrem, occ, done;
} Buf;

/* function prototype */
void *consumer(void *);

main(int argc, char **argv)
{
int ifd, ofd;
thread_t cons_thr;

/* check the command line arguments */
if (argc != 3)
```

```
        printf("Usage: %s <infile> <outfile>\n", argv[0]), exit(0);

/* open the input file for the producer to use */
if ((ifd = open(argv[1], O_RDONLY)) == -1)
        {
        fprintf(stderr, "Can't open file %s\n", argv[1]);
        exit(1);
        }

/* open the output file for the consumer to use */
if ((ofd = open(argv[2], O_WRONLY|O_CREAT, 0666)) == -1)
        {
        fprintf(stderr, "Can't open file %s\n", argv[2]);
        exit(1);
        }

/* zero the counters */
Buf.nextadd = Buf.nextrem = Buf.occ = Buf.done = 0;

/* set the thread concurrency to 2 so the producer and consumer can
   run concurrently */

thr_setconcurrency(2);

/* create the consumer thread */
thr_create(NULL, 0, consumer, (void *)ofd, NULL, &cons_thr);

/* the producer ! */
while (1) {

        /* lock the mutex */
        mutex_lock(&Buf.buflock);

        /* check to see if any buffers are empty */
        /* If not then wait for that condition to become true */

        while (Buf.occ == BUFCNT)
                cond_wait(&Buf.remdata, &Buf.buflock);
/* read from the file and put data into a buffer */
Buf.byteinbuf[Buf.nextadd] = read(ifd,Buf.buffer[Buf.nextadd],BUFSIZE);

/* check to see if done reading */
if (Buf.byteinbuf[Buf.nextadd] == 0) {

        /* lock the done lock */
        mutex_lock(&Buf.donelock);

        /* set the done flag and release the mutex lock */
        Buf.done = 1;

        mutex_unlock(&Buf.donelock);

        /* signal the consumer to start consuming */
        cond_signal(&Buf.adddata);

        /* release the buffer mutex */
        mutex_unlock(&Buf.buflock);

        /* leave the while looop */
        break;
        }

/* set the next buffer to fill */
Buf.nextadd = ++Buf.nextadd % BUFCNT;

/* increment the number of buffers that are filled */
Buf.occ++;

/* signal the consumer to start consuming */
cond_signal(&Buf.adddata);

/* release the mutex */
mutex_unlock(&Buf.buflock);
}
```

```
close(ifd);

/* wait for the consumer to finish */
thr_join(cons_thr, 0, NULL);

/* exit the program */
return(0);
}


/* The consumer thread */
void *consumer(void *arg)
{
int fd = (int) arg;

/* check to see if any buffers are filled or if the done flag is set */
while (1) {

        /* lock the mutex */
        mutex_lock(&Buf.buflock);

        if (!Buf.occ && Buf.done) {
           mutex_unlock(&Buf.buflock);
           break;
           }

        /* check to see if any buffers are filled */
        /* if not then wait for the condition to become true */

        while (Buf.occ == 0 && !Buf.done)
                cond_wait(&Buf.adddata, &Buf.buflock);

        /* write the data from the buffer to the file */
        write(fd, Buf.buffer[Buf.nextrem], Buf.byteinbuf[Buf.nextrem]);

        /* set the next buffer to write from */
        Buf.nextrem = ++Buf.nextrem % BUFCNT;
        /* decrement the number of buffers that are full */
        Buf.occ--;

        /* signal the producer that a buffer is empty */
        cond_signal(&Buf.remdata);

        /* release the mutex */
        mutex_unlock(&Buf.buflock);
        }

/* exit the thread */
thr_exit((void *)0);
}
```

31.7 A Socket Server

The socket server example uses threads to implement a ”standard” socket port server. The example shows how easy it is to use `thr_create()` calls in the place of `fork()` calls in existing programs.

A standard socket server should listen on a socket port and, when a message arrives, fork a process to service the request. Since a `fork()` system call would be used in a nonthreaded program, any communication between the parent and child would have to be done through some sort of interprocess communication.

We can replace the `fork()` call with a `thr_create()` call. Doing so offers a few advantages: `thr_create()` can create a thread much faster then a `fork()` could create a new process, and any communication between the *server* and the new thread can be done with common variables. This technique makes the implementation of the socket server much easier to understand and should also make it respond much faster to incoming requests.

The server program first sets up all the needed socket information. This is the basic setup for most socket servers. The server then enters an endless loop, waiting to service a socket port. When a message is sent to the socket port, the server wakes up and creates a new thread to handle the request. Notice that the server creates the new thread as a detached thread and also passes the socket descriptor as an argument to the new thread.

The newly created thread can then read or write, in any fashion it wants, to the socket descriptor that was passed to it. At this point the server could be creating a new thread or waiting for the next message to arrive. The key is that the server thread does not care what happens to the new thread after it creates it.

In our example, the created thread reads from the socket descriptor and then increments a global variable. This global variable keeps track of the number of requests that were made to the server. Notice that a mutex lock is used to protect access to the shared global variable. The lock is needed because many threads might try to increment the same variable at the same time. The mutex lock provides serial access to the shared variable. See how easy it is to share information among the new threads! If each of the threads were a process, then a significant effort would have to be made to share this information among the processes.

The client piece of the example sends a given number of messages to the server. This client code could also be run from different machines by multiple users, thus increasing the need for concurrency in the server process.

The source code to `soc_server.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <sys/uio.h>
#include <unistd.h>
#include <thread.h>

/* the TCP port that is used for this example */
#define TCP_PORT   6500

/* function prototypes and global variables */
void *do_chld(void *);
mutex_t lock;
int service_count;

main()
{
```

```
int  sockfd, newsockfd, clilen;
struct sockaddr_in cli_addr, serv_addr;
thread_t chld_thr;

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
fprintf(stderr,"server: can't open stream socket\n"), exit(0);

memset((char *) &serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(TCP_PORT);

if(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) <
0)
fprintf(stderr,"server: can't bind local address\n"), exit(0);

/* set the level of thread concurrency we desire */
thr_setconcurrency(5);

listen(sockfd, 5);

for(;;){
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);

if(newsockfd < 0)
fprintf(stderr,"server: accept error\n"), exit(0);

/* create a new thread to process the incomming request */
thr_create(NULL, 0, do_chld, (void *) newsockfd, THR_DETACHED,
&chld_thr);

/* the server is now free to accept another socket request */
}
return(0);
}
```

```
/*
This is the routine that is executed from a new thread
*/

void *do_chld(void *arg)
{
int  mysocfd = (int) arg;
char  data[100];
int  i;

printf("Child thread [%d]: Socket number = %d\n", thr_self(), mysocfd);

/* read from the given socket */
read(mysocfd, data, 40);

printf("Child thread [%d]: My data = %s\n", thr_self(), data);

/* simulate some processing */
for (i=0;i<1000000*thr_self();i++);

printf("Child [%d]: Done Processing...\n", thr_self());

/* use a mutex to update the global service counter */
mutex_lock(&lock);

service_count++;
mutex_unlock(&lock);

printf("Child thread [%d]: The total sockets served = %d\n", thr_self(), se

/* close the socket and exit this thread */
close(mysocfd);
thr_exit((void *)0);
}
```

31.8 Using Many Threads

This example that shows how easy it is to create many threads of execution in Solaris. Because of the lightweight nature of threads, it is possible to create literally thousands of threads. Most applications may not need a very large number of threads, but this example shows just how lightweight the threads can be.

We have said before that anything you can do with threads, you can do without them. This may be a case where it would be very hard to do without threads. If you have some spare time (and lots of memory), try implementing this program by using processes, instead of threads. If you try this, you will see why threads can have an advantage over processes.

This program takes as an argument the number of threads to create. Notice that all the threads are created with a user-defined stack size, which limits the amount of memory that the threads will need for execution. The stack size for a given thread can be hard to calculate, so some testing usually needs to be done to see if the chosen stack size will work. You may want to change the stack size in this program and see how much you can lower it before things stop working. The Solaris threads library provides the `thr_min_stack()` call, which returns the minimum allowed stack size. Take care when adjusting the size of a threads stack. A stack overflow can happen quite easily to a thread with a small stack.

After each thread is created, it blocks, waiting on a mutex variable. This mutex variable was locked before any of the threads were created, which prevents the threads from proceeding in their execution. When all of the threads have been created and the user presses Return, the mutex variable is unlocked, allowing all the threads to proceed.

After the main thread has created all the threads, it waits for user input and then tries to join all the threads. Notice that the `thr_join()` call does not care what thread it joins; it is just counting the number of joins it makes.

This example is rather trivial and does not serve any real purpose except to show that it is possible to create a lot of threads in one process. However, there are situations when many threads are needed in an application. An example might be a network port server, where a thread is created each time an incoming or outgoing request is made.

The source to `many_thr.c`:

```
#define _REENTRANT
```

```
#include <stdio.h>
#include <stdlib.h>
#include <thread.h>

/* function prototypes and global varaibles */
void *thr_sub(void *);
mutex_t lock;

main(int argc, char **argv)
{
int i, thr_count = 100;
char buf;

/* check to see if user passed an argument
   -- if so, set the number of threads to the value
      passed to the program */

if (argc == 2) thr_count = atoi(argv[1]);

printf("Creating %d threads...\n", thr_count);

/* lock the mutex variable -- this mutex is being used to
   keep all the other threads created from proceeding   */

mutex_lock(&lock);

/* create all the threads -- Note that a specific stack size is
   given.  Since the created threads will not use all of the
   default stack size, we can save memory by reducing the threads'
   stack size */

for (i=0;i<thr_count;i++) {
thr_create(NULL,2048,thr_sub,0,0,NULL);
}

printf("%d threads have been created and are running!\n", i);
printf("Press <return> to join all the threads...\n", i);
/* wait till user presses return, then join all the threads */
gets(&buf);

printf("Joining %d threads...\n", thr_count);

/* now unlock the mutex variable, to let all the threads proceed */
mutex_unlock(&lock);

/* join the threads */
for (i=0;i<thr_count;i++)
thr_join(0,0,0);

printf("All %d threads have been joined, exiting...\n", thr_count);
return(0);
}

/* The routine that is executed by the created threads */

void *thr_sub(void *arg)
{
/* try to lock the mutex variable -- since the main thread has
   locked the mutex before the threads were created, this thread
   will block until the main thread unlock the mutex */

mutex_lock(&lock);

printf("Thread %d is exiting...\n", thr_self());

/* unlock the mutex to allow another thread to proceed */
mutex_unlock(&lock);

/* exit the thread */
return((void *)0);
}
```

31.9 Real-time Thread Example

This example uses the Solaris real-time extensions to make a single bound thread within a process run in the real-time scheduling class. Using a thread in the real-time class is more desirable than running a whole process in the real-time class, because of the many problems that can arise with a process in a real-time state. For example, it would not be desirable for a process to perform any I/O or large memory operations while in realtime, because a real-time process has priority over system-related processes; if a real-time process requests a page fault, it can starve, waiting for the system to fault in a new page. We can limit this exposure by using threads to execute only the instructions that need to run in realtime.

Since this book does not cover the concerns that arise with real-time programming, we have included this code only as an example of how to promote a thread into the real-time class. You must be very careful when you use real-time threads in your applications. For more information on real-time programming, see the Solaris documentation.

This example should be safe from the pitfalls of real-time programs because of its simplicity. However, changing this code in any way could have adverse affects on your system.

The example creates a new thread from the main thread. This new thread is then promoted to the real-time class by looking up the real-time class ID and then setting a real-time priority for the thread. After the thread is running in realtime, it simulates some processing. Since a thread in the real-time class can have an infinite time quantum, the process is allowed to stay on a CPU as long as it likes. The time quantum is the amount of time a thread is allowed to stay running on a CPU. For the timesharing class, the time quantum (time-slice) is 1/100th of a second by default.

In this example, we set the time quantum for the real-time thread to infinity. That is, it can stay running as long as it likes; it will not be preempted or scheduled off the CPU. If you run this example on a UP machine, it will have the effect of stopping your system for a few seconds while the thread simulates its processing. The system does not actually stop, it is just working in the real-time thread. When the real-time thread finishes its processing, it exits and the system returns to normal.

Using real-time threads can be quite useful when you need an extremely high priority and response time but can also cause big problems if it not used properly. Also note that this example must be run as root or have root execute permissions.

The source to `rt_thr.c`:

```
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#include <string.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>

/* thread prototype */
void *rt_thread(void *);

main()
{

/* create the thread that will run in realtime */
thr_create(NULL, 0, rt_thread, 0, THR_DETACHED, 0);

/* loop here forever, this thread is the TS scheduling class */
while (1) {
printf("MAIN: In time share class... running\n");
sleep(1);
}

return(0);
}

/*
This is the routine that is called by the created thread
*/

void *rt_thread(void *arg)
{
pcinfo_t pcinfo;
pcparms_t pcparms;
int i;
/* let the main thread run for a bit */
sleep(4);

/* get the class ID for the real-time class */
strcpy(pcinfo.pc_clname, "RT");

if (priocntl(0, 0, PC_GETCID, (caddr_t)&pcinfo) == -1)
fprintf(stderr, "getting RT class id\n"), exit(1);

/* set up the real-time parameters */
pcparms.pc_cid = pcinfo.pc_cid;
((rtparms_t *)pcparms.pc_clparms)->rt_pri = 10;
((rtparms_t *)pcparms.pc_clparms)->rt_tqnsecs = 0;

/* set an infinite time quantum */
((rtparms_t *)pcparms.pc_clparms)->rt_tqsecs = RT_TQINF;

/* move this thread to the real-time scheduling class */
if (priocntl(P_LWPID, P_MYID, PC_SETPARMS, (caddr_t)&pcparms) == -1)
fprintf(stderr, "Setting RT mode\n"), exit(1);

/* simulate some processing */
for (i=0;i<100000000;i++);

printf("RT_THREAD: NOW EXITING...\n");
thr_exit((void *)0);
}
```

31.10 POSIX Cancellation

This example uses the POSIX thread cancellation capability to kill a thread that is no longer needed. Random termination of a thread can cause problems in threaded applications, because a thread may be holding a critical lock when it is terminated. Since the lock was help before the thread was terminated, another thread may deadlock, waiting for that same lock. The thread cancellation capability enables you to control when a thread can be terminated. The example also demonstrates the capabilities of the POSIX thread library in implementing a program that performs a multithreaded search.

This example simulates a multithreaded search for a given number by taking random guesses at a target number. The intent here is to simulate the same type of search that a database might execute. For example, a database might create threads to start searching for a data item; after some amount of time, one or more threads might return with the target data item.

If a thread guesses the number correctly, there is no need for the other threads to continue their search. This is where thread cancellation can help. The thread that finds the number first should cancel the other threads that are still searching for the item and then return the results of the search.

The threads involved in the search can call a cleanup function that can clean up the threads resources before it exits. In this case, the cleanup function prints the progress of the thread when it was cancelled.

The source to `posix_cancel.c`:

```
 #define _REENTRANT
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <pthread.h>

/* defines the number of searching threads */
#define NUM_THREADS 25

/* function prototypes */
void *search(void *);
void print_it(void *);

/* global variables */
pthread_t  threads[NUM_THREADS];
pthread_mutex_t lock;
int tries;

main()
{
```

```
int i;
int pid;

/* create a number to search for */
pid = getpid();

/* initialize the mutex lock */
pthread_mutex_init(&lock, NULL);
printf("Searching for the number = %d...\n", pid);

/* create the searching threads */
for (i=0;i<NUM_THREADS;i++)
pthread_create(&threads[i], NULL, search, (void *)pid);

/* wait for (join) all the searching threads */
for (i=0;i<NUM_THREADS;i++)
pthread_join(threads[i], NULL);

printf("It took %d tries to find the number.\n", tries);

/* exit this thread */
pthread_exit((void *)0);
}

/*
This is the cleanup function that is called when
the threads are cancelled
*/

void print_it(void *arg)
{
int *try = (int *) arg;
pthread_t tid;

/* get the calling thread's ID */
tid = pthread_self();

/* print where the thread was in its search when it was cancelled */
```

```
printf("Thread %d was canceled on its %d try.\n", tid, *try);
}

/*
This is the search routine that is executed in each thread
*/

void *search(void *arg)
{
int num = (int) arg;
int i=0, j;
pthread_t tid;

/* get the calling thread ID */
tid = pthread_self();

/* use the thread ID to set the seed for the random number generator */
srand(tid);

/* set the cancellation parameters --
   - Enable thread cancellation
   - Defer the action of the cancellation
*/

pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

/* push the cleanup routine (print_it) onto the thread
   cleanup stack.  This routine will be called when the
   thread is cancelled.  Also note that the pthread_cleanup_push
   call must have a matching pthread_cleanup_pop call.  The
   push and pop calls MUST be at the same lexical level
   within the code */

/* pass address of `i' since the current value of `i' is not
   the one we want to use in the cleanup function */

pthread_cleanup_push(print_it, (void *)&i);
```

```
/* loop forever */
while (1) {
i++;

/* does the random number match the target number? */
if (num == rand()) {

/* try to lock the mutex lock --
                   if locked, check to see if the thread has been cancelled
   if not locked then continue */

while (pthread_mutex_trylock(&lock) == EBUSY)
pthread_testcancel();

/* set the global variable for the number of tries */

    tries = i;

    printf("thread %d found the number!\n", tid);

/* cancel all the other threads */
    for (j=0;j<NUM_THREADS;j++)
if (threads[j] != tid) pthread_cancel(threads[j]);

/* break out of the while loop */
break;
   }

/* every 100 tries check to see if the thread has been cancelled
           if the thread has not been cancelled then yield the thread's
   LWP to another thread that may be able to run */

if (i%100 == 0) {
pthread_testcancel();
sched_yield();
}
}
```

```
/* The only way we can get here is when the thread breaks out
   of the while loop.  In this case the thread that makes it here
   has found the number we are looking for and does not need to run
   the thread cleanup function.  This is why the pthread_cleanup_pop
   function is called with a 0 argument; this will pop the cleanup
   function off the stack without executing it */

pthread_cleanup_pop(0);
return((void *)0);
}
```

31.11 Software Race Condition

This example shows a trivial software race condition. A software race condition occurs when the execution of a program is affected by the order and timing of a threads execution. Most software race conditions can be alleviated by using synchronization variables to control the threads' timing and access of shared resources. If a program depends on order of execution, then threading that program may not be a good solution, because the order in which threads execute is nondeterministic.

In the example, `thr_continue()` and `thr_suspend()` calls continue and suspend a given thread, respectively. Although both of these calls are valid, use caution when implementing them. It is very hard to determine where a thread is in its execution. Because of this, you may not be able to tell where the thread will suspend when the call to `thr_suspend()` is made. This behavior can cause problems in threaded code if not used properly.

The following example uses `thr_continue()` and `thr_suspend()` to try to control when a thread starts and stops. The example looks trivial, but, as you will see, can cause a big problem.

Do you see the problem? If you guessed that the program would eventually suspend itself, you were correct! The example attempts to flip-flop between the main thread and a subroutine thread. Each thread continues the other thread and then suspends itself.

Thread A continues thread B and then suspends thread A; now the continued thread B can continue thread A and then suspend itself. This should continue back and forth all day long, right? Wrong! We can't guarantee that each thread will continue the other thread and then suspend itself in one atomic action, so a software race condition could be created. Calling `thr_continue()` on a running thread and calling `thr_suspend()` on a suspended thread has no effect, so we don't know if a thread is already running or suspended.

If thread A continues thread B and if between the time thread A suspends itself, thread B continues thread A, then both of the threads will call `thr_suspend()`. This is the race condition in this program that will cause the whole process to become suspended.

It is very hard to use these calls, because you never really know the state of a thread. If you don't know exactly where a thread is in its execution, then you don't know what locks it holds and where it will stop when you suspend it.

The source to `sw_race.c`

31.12 `Tgrep`: Threadeds version of UNIX `grep`

`Tgrep` is a multi-threaded version of `grep`. `Tgrep` supports all but the -w (word search) options of the normal `grep` command, and a few options that are only avaliable to `Tgrep`. The real change from `grep`, is that `Tgrep` will recurse down through sub-directories and search all files for the target string. `Tgrep` searches files like the following command:

```
find <start path> -name "<file/directory pattern>" -exec \ (Line wrapped)
        grep <options> <target> /dev/null {} \;
```

An example of this would be (run from this `Tgrep` directory)

```
% find . -exec grep thr_create /dev/null {} \;
./Solaris/main.c:  if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
./Solaris/main.c:          err = thr_create(NULL,0,cascade,(void *)work,
./Solaris/main.c:           err = thr_create(NULL,0,search_thr,(void *)work
%
Running the same command with timex:
real        4.26
user        0.64
sys         2.81
```

The same search run with `Tgrep` would be

```
% {\tt Tgrep} thr_create
./Solaris/main.c:  if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
./Solaris/main.c:          err = thr_create(NULL,0,cascade,(void *)work,
./Solaris/main.c:           err = thr_create(NULL,0,search_thr,(void *)work,
%
Running the same command with timex:
real        0.79
user        0.62
sys         1.50
```

`Tgrep` gets the results almost four times faster. The numbers above where gathered on a SS20 running 5.5 (build 18) with 4 50MHz CPUs.

You can also filter the files that you want `Tgrep` to search like you can with find. The next two commands do the same thing, just `Tgrep` gets it done faster.

```
find . -name "*.c" -exec grep thr_create /dev/null {} \;
and
{\tt Tgrep} -p '.*\.c$' thr_create
```

The -p option will allow `Tgrep` to search only files that match the "regular expression" file pattern string. This option does NOT use shell expression, so to stop `Tgrep` from seeing a file named foobar.cỹou must add the "$" meta character to the pattern and escape the real "." character.

Some of the other `Tgrep` only options are -r, -C, -P, -e, -B, -S and -Z. The -r option stops `Tgrep` from searching any sub-directories, in other words, search only the local directory, but -l was taken. The -C option will search for and print "continued" lines like you find in Makefile. Note the differences in the results of grep and `Tgrep` run in the current directory.

The `Tgrep` output prints the continued lines that ended with the "c̈haracter. In the case of grep I would not have seen the three values assigned to SUBDIRS, but `Tgrep` shows them to me (Common, Solaris, Posix).

The -P option I use when I am sending the output of a long search to a file and want to see the "progress" of the search. The -P option will print a "." (dot) on stderr for every file (or groups of files depending on the value of the -P argument) `Tgrep` searches.

The -e option will change the way Tgrep uses the target string. Tgrep uses two different patter matching systems. The first (with out the -e option) is a literal string match call Boyer-Moore. If the -e option is used, then a MT-Safe PD version of regular expression is used to search for the target string as a regexp with meta characters in it. The regular expression method is slower, but Tgrep needed the functionality. The -Z option will print help on the meta characters Tgrep uses.

The -B option tells Tgrep to use the value of the environment variable called TGLIMIT to limit the number of threads it will use during a search. This option has no affect if TGLIMIT is not set. Tgrep can "eat" a system alive, so the -B option was a way to run Tgrep on a system with out having other users scream at you.

The last new option is -S. If you want to see how things went while Tgrep was searching, you can use this option to print statistic about the number of files, lines, bytes, matches, threads created, etc.

Here is an example of the -S options output. (again run in the current directory)

```
% {\tt Tgrep} -S zimzap

----------------- {\tt Tgrep} Stats. --------------------
Number of directories searched:          7
Number of files searched:                37
Number of lines searched:                9504
Number of matching lines to target:      0
Number of cascade threads created:       7
Number of search threads created:        20
Number of search threads from pool:      17
Search thread pool hit rate:             45.95%
Search pool overall size:                20
Search pool size limit:                  58
Number of search threads destroyed:      0
Max # of threads running concurrenly:    20
Total run time, in seconds.              1
Work stopped due to no FD's:  (058)      0 Times, 0.00%
Work stopped due to no work on Q:        19 Times, 43.18%
Work stopped due to TGLIMITS: (Unlimited) 0 Times, 0.00%
----------------------------------------------------
```

%

For more information on the usage and options, see the man page Tgrep
The Tgrep.c source code is:

```
/* Copyright (c) 1993, 1994  Ron Winacott                              */
/* This program may be used, copied, modified, and redistributed freely */
/* for ANY purpose, so long as this notice remains intact.             */

#define _REENTRANT

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
#include <sys/stat.h>
#ifdef __sparc
#include <note.h> /* warlock/locklint */
#else
#define NOTE(s)
#endif
#include <dirent.h>
#include <fcntl.h>
#include <sys/uio.h>
#include <thread.h>
#include <synch.h>

#include "version.h"
#include "pmatch.h"
#include "debug.h"
```

```
#define PATH_MAX 1024 /* max # of characters in a path name */
#define HOLD_FDS                6  /* stdin,out,err and a buffer */
#define UNLIMITED               99999 /* The default tglimit */
#define MAXREGEXP               10  /* max number of -e options */

#define FB_BLOCK                0x00001
#define FC_COUNT                0x00002
#define FH_HOLDNAME             0x00004
#define FI_IGNCASE              0x00008
#define FL_NAMEONLY             0x00010
#define FN_NUMBER               0x00020
#define FS_NOERROR              0x00040
#define FV_REVERSE              0x00080
#define FW_WORD                 0x00100
#define FR_RECUR                0x00200
#define FU_UNSORT               0x00400
#define FX_STDIN                0x00800
#define TG_BATCH                0x01000
#define TG_FILEPAT              0x02000
#define FE_REGEXP               0x04000
#define FS_STATS                0x08000
#define FC_LINE                 0x10000
#define TG_PROGRESS             0x20000

#define FILET                   1
#define DIRT                    2
#define ALPHASIZ         128

/*
 * New data types
 */

typedef struct work_st {
    char             *path;
    int              tp;
    struct work_st   *next;
} work_t;
```

```
typedef struct out_st {
    char                *line;
    int                 line_count;
    long                byte_count;
    struct out_st       *next;
} out_t;

typedef struct bm_pattern {     /* Boyer - Moore pattern                */
        short            p_m;          /* length of pattern string     */
        short            p_r[ALPHASIZ]; /* "r" vector                  */
        short           *p_R;          /* "R" vector                   */
        char            *p_pat;        /* pattern string               */
} BM_PATTERN;


/*
 * Prototypes
 */

/* bmpmatch.c */
extern BM_PATTERN *bm_makepat(char *);
extern char *bm_pmatch(BM_PATTERN *, register char *);
extern void bm_freepat(BM_PATTERN *);
/* pmatch.c */
extern char *pmatch(register PATTERN *, register char *, int *);
extern PATTERN *makepat(char *string, char *);
extern void freepat(register PATTERN *);
extern void printpat(PATTERN *);

#include "proto.h"  /* function prototypes of main.c */

void *SigThread(void *arg);
void sig_print_stats(void);

/*
 * Global data
 */
```

```
BM_PATTERN      *bm_pat;  /* the global target read only after main */
NOTE(READ_ONLY_DATA(bm_pat))

PATTERN         *pm_pat[MAXREGEXP];  /* global targets read only for pmatch
NOTE(READ_ONLY_DATA(pm_pat))

mutex_t global_count_lk;
int     global_count = 0;
NOTE(MUTEX_PROTECTS_DATA(global_count_lk, global_count))
NOTE(DATA_READABLE_WITHOUT_LOCK(global_count))  /* see prnt_stats() */

work_t  *work_q = NULL;
cond_t  work_q_cv;
mutex_t work_q_lk;
int     all_done = 0;
int     work_cnt = 0;
int     current_open_files = 0;
int     tglimit = UNLIMITED;    /* if -B limit the number of threads */
NOTE(MUTEX_PROTECTS_DATA(work_q_lk, work_q all_done work_cnt \
 current_open_files tglimit))

work_t  *search_q = NULL;
mutex_t search_q_lk;
cond_t  search_q_cv;
int     search_pool_cnt = 0;    /* the count in the pool now */
int     search_thr_limit = 0;   /* the max in the pool */
NOTE(MUTEX_PROTECTS_DATA(search_q_lk, search_q search_pool_cnt))
NOTE(DATA_READABLE_WITHOUT_LOCK(search_pool_cnt)) /* see prnt_stats() */
NOTE(READ_ONLY_DATA(search_thr_limit))

work_t *cascade_q = NULL;
mutex_t cascade_q_lk;
cond_t  cascade_q_cv;
int cascade_pool_cnt = 0;
int     cascade_thr_limit = 0;
NOTE(MUTEX_PROTECTS_DATA(cascade_q_lk, cascade_q cascade_pool_cnt))
NOTE(DATA_READABLE_WITHOUT_LOCK(cascade_pool_cnt))  /* see prnt_stats() */
NOTE(READ_ONLY_DATA(cascade_thr_limit))

int     running = 0;
mutex_t running_lk;
NOTE(MUTEX_PROTECTS_DATA(running_lk, running))

sigset_t set, oldset;
NOTE(READ_ONLY_DATA(set oldset))

mutex_t stat_lk;
time_t  st_start = 0;
int     st_dir_search = 0;
int     st_file_search = 0;
int     st_line_search = 0;
int     st_cascade = 0;
int st_cascade_pool = 0;
int     st_cascade_destroy = 0;
int     st_search = 0;
int     st_pool = 0;
int     st_maxrun = 0;
int     st_worknull = 0;
int     st_workfds = 0;
int     st_worklimit = 0;
int     st_destroy = 0;
NOTE(MUTEX_PROTECTS_DATA(stat_lk, st_start st_dir_search st_file_search \
 st_line_search st_cascade st_cascade_pool \
 st_cascade_destroy st_search st_pool st_maxrun \
 st_worknull st_workfds st_worklimit st_destroy))

int     progress_offset = 1;
NOTE(READ_ONLY_DATA(progress_offset))

mutex_t output_print_lk;
/* output_print_lk used to print multi-line output only */
int     progress = 0;
NOTE(MUTEX_PROTECTS_DATA(output_print_lk, progress))

unsigned int    flags = 0;
```

```
int     regexp_cnt = 0;
char    *string[MAXREGEXP];
int     debug = 0;
int     use_pmatch = 0;
char    file_pat[255];  /* file patten match */
PATTERN *pm_file_pat; /* compiled file target string (pmatch()) */
NOTE(READ_ONLY_DATA(flags regexp_cnt string debug use_pmatch \
    file_pat pm_file_pat))


/*
 * Locking ording.
 */
NOTE(LOCK_ORDER(output_print_lk stat_lk))

/*
 * Main: This is where the fun starts
 */

int
main(int argc, char **argv)
{
    int         c,out_thr_flags;
    long        max_open_files = 0l, ncpus = 0l;
    extern int  optind;
    extern char *optarg;
    NOTE(READ_ONLY_DATA(optind optarg))
    int         prio = 0;
    struct stat sbuf;
    thread_t    tid,dtid;
    void        *status;
    char        *e = NULL, *d = NULL; /* for debug flags */
    int         debug_file = 0;
    int         err = 0, i = 0, pm_file_len = 0;
    work_t      *work;
    int         restart_cnt = 10;

    flags = FR_RECUR;  /* the default */
    thr_setprio(thr_self(),127);  /* set me up HIGH */
    while ((c = getopt(argc, argv, "d:e:bchilnsvwruf:p:BCSZzHP:")) != EOF) {
        switch (c) {
#ifdef DEBUG
        case 'd':
            debug = atoi(optarg);
            if (debug == 0)
                debug_usage();

            d = optarg;
            fprintf(stderr,"tgrep: Debug on at level(s) ");
            while (*d) {
                for (i=0; i<9; i++)
                    if (debug_set[i].level == *d) {
                        debug_levels |= debug_set[i].flag;
                        fprintf(stderr,"%c ",debug_set[i].level);
                        break;
                    }
                d++;
            }
            fprintf(stderr,"\n");
            break;
        case 'f':
    debug_file = atoi(optarg);
    break;
#endif      /* DEBUG */
        case 'B':
            flags |= TG_BATCH;
    if ((e = getenv("TGLIMIT"))) {
tglimit = atoi(e);
    }
    else {
if (!(flags & FS_NOERROR))  /* order dependent! */
    fprintf(stderr,"env TGLIMIT not set, overriding -B\n");
flags &= ~TG_BATCH;
    }
            break;
```

```
case 'p':
    flags |= TG_FILEPAT;
    strcpy(file_pat,optarg);
    pm_file_pat = makepat(file_pat,NULL);
    break;
case 'P':
    flags |= TG_PROGRESS;
    progress_offset = atoi(optarg);
    break;
case 'S': flags |= FS_STATS;    break;
case 'b': flags |= FB_BLOCK;    break;
case 'c': flags |= FC_COUNT;    break;
case 'h': flags |= FH_HOLDNAME; break;
case 'i': flags |= FI_IGNCASE;  break;
case 'l': flags |= FL_NAMEONLY; break;
case 'n': flags |= FN_NUMBER;   break;
case 's': flags |= FS_NOERROR;  break;
case 'v': flags |= FV_REVERSE;  break;
case 'w': flags |= FW_WORD;     break;
case 'r': flags &= ~FR_RECUR;   break;
case 'C': flags |= FC_LINE;     break;
case 'e':
    if (regexp_cnt == MAXREGEXP) {
        fprintf(stderr,"Max number of regexp's (%d) exceeded!\n",
                MAXREGEXP);
        exit(1);
    }
    flags |= FE_REGEXP;
    if ((string[regexp_cnt] =(char *)malloc(strlen(optarg)+1))==NUL
        fprintf(stderr,"tgrep: No space for search string(s)\n");
        exit(1);
    }
    memset(string[regexp_cnt],0,strlen(optarg)+1);
    strcpy(string[regexp_cnt],optarg);
    regexp_cnt++;
    break;
case 'z':
case 'Z': regexp_usage();
```

```
            break;
        case 'H':
        case '?':
        default : usage();
        }
    }

    if (!(flags & FE_REGEXP)) {
       if (argc - optind < 1) {
            fprintf(stderr,"tgrep: Must supply a search string(s) "
                    "and file list or directory\n");
            usage();
        }
        if ((string[0]=(char *)malloc(strlen(argv[optind])+1))==NULL){
            fprintf(stderr,"tgrep: No space for search string(s)\n");
            exit(1);
        }
        memset(string[0],0,strlen(argv[optind])+1);
        strcpy(string[0],argv[optind]);
        regexp_cnt=1;
        optind++;
    }

    if (flags & FI_IGNCASE)
        for (i=0; i<regexp_cnt; i++)
            uncase(string[i]);

#ifdef __lock_lint
    /*
    ** This is NOT somthing you really want to do. This
    ** function calls are here ONLY for warlock/locklint !!!
    */
    pm_pat[i] = makepat(string[i],NULL);
    bm_pat = bm_makepat(string[0]);
    bm_freepat(bm_pat);  /* stop it from becomming a root */
#else
    if (flags & FE_REGEXP) {
        for (i=0; i<regexp_cnt; i++)
```

```
            pm_pat[i] = makepat(string[i],NULL);
        use_pmatch = 1;
    }
    else {
        bm_pat = bm_makepat(string[0]); /* only one allowed */
    }
#endif

    flags |= FX_STDIN;

    max_open_files = sysconf(_SC_OPEN_MAX);
    ncpus = sysconf(_SC_NPROCESSORS_ONLN);
    if ((max_open_files - HOLD_FDS - debug_file) < 1) {
        fprintf(stderr,"tgrep: You MUST have at lest ONE fd "
                "that can be used, check limit (>10)\n");
        exit(1);
    }
    search_thr_limit = max_open_files - HOLD_FDS - debug_file;
    cascade_thr_limit = search_thr_limit / 2;
    /* the number of files that can by open */
    current_open_files = search_thr_limit;

    mutex_init(&stat_lk,USYNC_THREAD,"stat");
    mutex_init(&global_count_lk,USYNC_THREAD,"global_cnt");
    mutex_init(&output_print_lk,USYNC_THREAD,"output_print");
    mutex_init(&work_q_lk,USYNC_THREAD,"work_q");
    mutex_init(&running_lk,USYNC_THREAD,"running");
    cond_init(&work_q_cv,USYNC_THREAD,"work_q");
    mutex_init(&search_q_lk,USYNC_THREAD,"search_q");
    cond_init(&search_q_cv,USYNC_THREAD,"search_q");
    mutex_init(&cascade_q_lk,USYNC_THREAD,"cascade_q");
    cond_init(&cascade_q_cv,USYNC_THREAD,"cascade_q");

    if ((argc == optind) && ((flags & TG_FILEPAT) || (flags & FR_RECUR))) {
        add_work(".",DIRT);
        flags = (flags & ~FX_STDIN);
    }
    for ( ; optind < argc; optind++) {
        restart_cnt = 10;
        flags = (flags & ~FX_STDIN);
      STAT_AGAIN:
        if (stat(argv[optind], &sbuf)) {
            if (errno == EINTR) { /* try again !, restart */
                if (--restart_cnt)
                    goto STAT_AGAIN;
            }
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't stat file/dir %s, %s\n",
                        argv[optind], strerror(errno));
            continue;
        }
        switch (sbuf.st_mode & S_IFMT) {
        case S_IFREG :
            if (flags & TG_FILEPAT) {
                if (pmatch(pm_file_pat, argv[optind], &pm_file_len))
    add_work(argv[optind],FILET);
            }
            else {
                add_work(argv[optind],FILET);
            }
            break;
        case S_IFDIR :
            if (flags & FR_RECUR) {
                add_work(argv[optind],DIRT);
            }
            else {
                if (!(flags & FS_NOERROR))
                    fprintf(stderr,"tgrep: Can't search directory %s, "
                            "-r option is on. Directory ignored.\n",
                            argv[optind]);
            }
            break;
        }
    }

    NOTE(COMPETING_THREADS_NOW)  /* we are goinf threaded */
    if (flags & FS_STATS) {
mutex_lock(&stat_lk);
        st_start = time(NULL);
mutex_unlock(&stat_lk);
#ifdef SIGNAL_HAND
/*
** setup the signal thread so the first call to SIGINT will
** only print stats, the second will interupt.
*/
sigfillset(&set);
thr_sigsetmask(SIG_SETMASK, &set, &oldset);
if (thr_create(NULL,0,SigThread,NULL,THR_DAEMON,NULL)) {
    thr_sigsetmask(SIG_SETMASK,&oldset,NULL);
    fprintf(stderr,"SIGINT for stats NOT setup\n");
}
thr_yield(); /* give the other thread time */
#endif /* SIGNAL_HAND */
    }

    thr_setconcurrency(3);

    if (flags & FX_STDIN) {
        fprintf(stderr,"tgrep: stdin option is not coded at this time\n");
        exit(0);                        /* XXX Need to fix this SOON */
        search_thr(NULL);  /* NULL is not understood in search_thr() */
        if (flags & FC_COUNT) {
            mutex_lock(&global_count_lk);
            printf("%d\n",global_count);
            mutex_unlock(&global_count_lk);
        }
        if (flags & FS_STATS) {
    mutex_lock(&stat_lk);
            prnt_stats();
    mutex_unlock(&stat_lk);
}
        exit(0);
    }
    mutex_lock(&work_q_lk);
    if (!work_q) {
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: No files to search.\n");
        exit(0);
    }
    mutex_unlock(&work_q_lk);

    DP(DLEVEL1,("Starting to loop through the work_q for work\n"));

    /* OTHER THREADS ARE RUNNING */
    while (1) {
        mutex_lock(&work_q_lk);
        while ((work_q == NULL || current_open_files == 0 || tglimit <= 0) &&
               all_done == 0) {
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                if (work_q == NULL)
                    st_worknull++;
                if (current_open_files == 0)
                    st_workfds++;
                if (tglimit <= 0)
                    st_worklimit++;
                mutex_unlock(&stat_lk);
            }
            cond_wait(&work_q_cv,&work_q_lk);
        }
        if (all_done != 0) {
            mutex_unlock(&work_q_lk);
            DP(DLEVEL1,("All_done was set to TRUE\n"));
            goto OUT;
        }
        work = work_q;
        work_q = work->next;  /* maybe NULL */
        work->next = NULL;
        current_open_files--;
        mutex_unlock(&work_q_lk);
        tid = 0;
        switch (work->tp) {
        case DIRT:
    mutex_lock(&cascade_q_lk);
    if (cascade_pool_cnt) {
if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_cascade_pool++;
                    mutex_unlock(&stat_lk);
                }
work->next = cascade_q;
cascade_q = work;
cond_signal(&cascade_q_cv);
                mutex_unlock(&cascade_q_lk);
                DP(DLEVEL2,("Sent work to cascade pool thread\n"));
    }
    else {
mutex_unlock(&cascade_q_lk);
err = thr_create(NULL,0,cascade,(void *)work,
 THR_DETACHED|THR_DAEMON|THR_NEW_LWP
 ,&tid);
DP(DLEVEL2,("Sent work to new cascade thread\n"));
thr_setprio(tid,64);  /* set cascade to middle */
if (flags & FS_STATS) {
    mutex_lock(&stat_lk);
    st_cascade++;
    mutex_unlock(&stat_lk);
}
    }
            break;
        case FILET:
            mutex_lock(&search_q_lk);
            if (search_pool_cnt) {
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_pool++;
                    mutex_unlock(&stat_lk);
                }
                work->next = search_q;  /* could be null */
                search_q = work;
                cond_signal(&search_q_cv);
                mutex_unlock(&search_q_lk);
                DP(DLEVEL2,("Sent work to search pool thread\n"));
            }
            else {
                mutex_unlock(&search_q_lk);
                err = thr_create(NULL,0,search_thr,(void *)work,
                                 THR_DETACHED|THR_DAEMON|THR_NEW_LWP
                                 ,&tid);
                thr_setprio(tid,0);  /* set search to low */
                DP(DLEVEL2,("Sent work to new search thread\n"));
                if (flags & FS_STATS) {
                    mutex_lock(&stat_lk);
                    st_search++;
                    mutex_unlock(&stat_lk);
                }
            }
            break;
        default:
            fprintf(stderr,"tgrep: Internal error, work_t->tp no valid\n");
            exit(1);
        }
        if (err) {  /* NEED TO FIX THIS CODE. Exiting is just wrong */
            fprintf(stderr,"Cound not create new thread!\n");
            exit(1);
        }
    }

 OUT:
    if (flags & TG_PROGRESS) {
        if (progress)
            fprintf(stderr,".\n");
        else
            fprintf(stderr,"\n");
    }
    /* we are done, print the stuff. All other threads ar parked */
    if (flags & FC_COUNT) {
        mutex_lock(&global_count_lk);
        printf("%d\n",global_count);
        mutex_unlock(&global_count_lk);
    }
    if (flags & FS_STATS)
        prnt_stats();
    return(0); /* should have a return from main */
}


/*
 * Add_Work: Called from the main thread, and cascade threads to add file
 * and directory names to the work Q.
 */
int
add_work(char *path,int tp)
{
    work_t      *wt,*ww,*wp;

    if ((wt = (work_t *)malloc(sizeof(work_t))) == NULL)
        goto ERROR;
    if ((wt->path = (char *)malloc(strlen(path)+1)) == NULL)
        goto ERROR;

    strcpy(wt->path,path);
    wt->tp = tp;
    wt->next = NULL;
    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        if (wt->tp == DIRT)
            st_dir_search++;
        else
            st_file_search++;
        mutex_unlock(&stat_lk);
    }
    mutex_lock(&work_q_lk);
    work_cnt++;
    wt->next = work_q;
    work_q = wt;
    cond_signal(&work_q_cv);
    mutex_unlock(&work_q_lk);
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Could not add %s to work queue. Ignored\n",
                path);
    return(-1);
}

/*
 * Search thread: Started by the main thread when a file name is found
 * on the work Q to be serached. If all the needed resources are ready
 * a new search thread will be created.
 */
void *
search_thr(void *arg) /* work_t *arg */
{
    FILE        *fin;
    char        fin_buf[(BUFSIZ*4)];  /* 4 Kbytes */
    work_t      *wt,std;
    int         line_count;
    char        rline[128];
    char        cline[128];
    char        *line;
    register char *p,*pp;
    int            pm_len;
    int         len = 0;
    long        byte_count;
    long        next_line;
    int         show_line;  /* for the -v option */
    register int slen,plen,i;
    out_t       *out = NULL;    /* this threads output list */

    thr_setprio(thr_self(),0);  /* set search to low */
    thr_yield();
    wt = (work_t *)arg; /* first pass, wt is passed to use. */

    /* len = strlen(string);*/  /* only set on first pass */

    while (1) {  /* reuse the search threads */
        /* init all back to zero */
        line_count = 0;
        byte_count = 0l;
        next_line = 0l;
        show_line = 0;

        mutex_lock(&running_lk);
        running++;
        mutex_unlock(&running_lk);
        mutex_lock(&work_q_lk);
        tglimit--;
        mutex_unlock(&work_q_lk);
        DP(DLEVEL5,("searching file (STDIO) %s\n",wt->path));

        if ((fin = fopen(wt->path,"r")) == NULL) {
            if (!(flags & FS_NOERROR)) {
                fprintf(stderr,"tgrep: %s. File \"%s\" not searched.\n",
                        strerror(errno),wt->path);
            }
            goto ERROR;
        }
        setvbuf(fin,fin_buf,_IOFBF,(BUFSIZ*4));  /* XXX */
        DP(DLEVEL5,("Search thread has opened file %s\n",wt->path));
        while ((fgets(rline,127,fin)) != NULL) {
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_line_search++;
                mutex_unlock(&stat_lk);
            }
            slen = strlen(rline);
            next_line += slen;
            line_count++;
            if (rline[slen-1] == '\n')
                rline[slen-1] = '\0';
            /*
            ** If the uncase flag is set, copy the read in line (rline)
            ** To the uncase line (cline) Set the line pointer to point at
            ** cline.
            ** If the case flag is NOT set, then point line at rline.
            ** line is what is compared, rline is waht is printed on a
            ** match.
            */
            if (flags & FI_IGNCASE) {
                strcpy(cline,rline);
                uncase(cline);
                line = cline;
            }
            else {
                line = rline;
            }
            show_line = 1;  /* assume no match, if -v set */
            /* The old code removed */
            if (use_pmatch) {
                for (i=0; i<regexp_cnt; i++) {
                    if (pmatch(pm_pat[i], line, &pm_len)) {
                        if (!(flags & FV_REVERSE)) {
                            add_output_local(&out,wt,line_count,
                                             byte_count,rline);
                            continue_line(rline,fin,out,wt,
                                          &line_count,&byte_count);
                        }
                        else {
                            show_line = 0;
                        } /* end of if -v flag if / else block */
                        /*
                        ** if we get here on ANY of the regexp targets
                        ** jump out of the loop, we found a single
                        ** match so, do not keep looking!
                        ** If name only, do not keep searcthing the same
** file, we found a single match, so close the file,
** print the file name and move on to the next file.
                        */
                        if (flags & FL_NAMEONLY)
                            goto OUT_OF_LOOP;
                        else
                            goto OUT_AND_DONE;
                    } /* end found a match if block */
                } /* end of the for pat[s] loop */
            }
            else {
                if (bm_pmatch( bm_pat, line)) {
                    if (!(flags & FV_REVERSE)) {
                        add_output_local(&out,wt,line_count,byte_count,rlin
                        continue_line(rline,fin,out,wt,
                                      &line_count,&byte_count);
                    }
                    else {
                        show_line = 0;
                    }
                    if (flags & FL_NAMEONLY)
                        goto OUT_OF_LOOP;
                }
            }
          OUT_AND_DONE:
            if ((flags & FV_REVERSE) && show_line) {
                add_output_local(&out,wt,line_count,byte_count,rline);
                show_line = 0;
            }
            byte_count = next_line;
        }
      OUT_OF_LOOP:
        fclose(fin);
        /*
        ** The search part is done, but before we give back the FD,
        ** and park this thread in the search thread pool, print the
        ** local output we have gathered.
        */
        print_local_output(out,wt);  /* this also frees out nodes */
        out = NULL; /* for the next time around, if there is one */
    ERROR:
        DP(DLEVEL5,("Search done for %s\n",wt->path));
        free(wt->path);
        free(wt);

        notrun();
        mutex_lock(&search_q_lk);
        if (search_pool_cnt > search_thr_limit) {
            mutex_unlock(&search_q_lk);
            DP(DLEVEL5,("Search thread exiting\n"));
            if (flags & FS_STATS) {
                mutex_lock(&stat_lk);
                st_destroy++;
                mutex_unlock(&stat_lk);
            }
            return(0);
        }
        else {
            search_pool_cnt++;
            while (!search_q)
                cond_wait(&search_q_cv,&search_q_lk);
            search_pool_cnt--;
            wt = search_q;  /* we have work to do! */
            if (search_q->next)
                search_q = search_q->next;
            else
                search_q = NULL;
            mutex_unlock(&search_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Continue line: Speacial case search with the -C flag set. If you are
 * searching files like Makefiles, some lines may have escape char's to
 * contine the line on the next line. So the target string can be found, but
 * no data is displayed. This function continues to print the escaped line
 * until there are no more "\" chars found.
 */
int
continue_line(char *rline, FILE *fin, out_t *out, work_t *wt,
      int *lc, long *bc)
{
    int len;
    int cnt = 0;
    char *line;
    char nline[128];

    if (!(flags & FC_LINE))
        return(0);

    line = rline;
  AGAIN:
    len = strlen(line);
    if (line[len-1] == '\\') {
        if ((fgets(nline,127,fin)) == NULL) {
            return(cnt);
        }
        line = nline;
        len = strlen(line);
        if (line[len-1] == '\n')
            line[len-1] = '\0';
        *bc = *bc + len;
        *lc++;
        add_output_local(&out,wt,*lc,*bc,line);
        cnt++;
        goto AGAIN;
    }
    return(cnt);
}

/*
 * cascade: This thread is started by the main thread when directory names
 * are found on the work Q. The thread reads all the new file, and director
 * names from the directory it was started when and adds the names to the
 * work Q. (it finds more work!)
 */
void *
cascade(void *arg)  /* work_t *arg */
{
    char        fullpath[1025];
    int         restart_cnt = 10;
    DIR         *dp;

    char        dir_buf[sizeof(struct dirent) + PATH_MAX];
    struct dirent *dent = (struct dirent *)dir_buf;
    struct stat   sbuf;
    char        *fpath;
    work_t      *wt;
    int         fl = 0, dl = 0;
    int         pm_file_len = 0;

    thr_setprio(thr_self(),64);  /* set search to middle */
    thr_yield();  /* try toi give control back to main thread */
    wt = (work_t *)arg;

    while(1) {
fl = 0;
dl = 0;
restart_cnt = 10;
pm_file_len = 0;

mutex_lock(&running_lk);
running++;
mutex_unlock(&running_lk);
mutex_lock(&work_q_lk);
tglimit--;
mutex_unlock(&work_q_lk);

if (!wt) {
    if (!(flags & FS_NOERROR))
fprintf(stderr,"tgrep: Bad work node passed to cascade\n");
```

```
    goto DONE;
}
fpath = (char *)wt->path;
if (!fpath) {
    if (!(flags & FS_NOERROR))
fprintf(stderr,"tgrep: Bad path name passed to cascade\n");
    goto DONE;
}
DP(DLEVEL3,("Cascading on %s\n",fpath));
if (( dp = opendir(fpath)) == NULL) {
    if (!(flags & FS_NOERROR))
fprintf(stderr,"tgrep: Can't open dir %s, %s. Ignored.\n",
fpath,strerror(errno));
    goto DONE;
}
while ((readdir_r(dp,dent)) != NULL) {
    restart_cnt = 10;  /* only try to restart the interupted 10 X */

    if (dent->d_name[0] == '.') {
if (dent->d_name[1] == '.' && dent->d_name[2] == '\0')
    continue;
if (dent->d_name[1] == '\0')
    continue;
    }

    fl = strlen(fpath);
    dl = strlen(dent->d_name);
    if ((fl + 1 + dl) > 1024) {
fprintf(stderr,"tgrep: Path %s/%s is too long. "
"MaxPath = 1024\n",
fpath, dent->d_name);
continue;  /* try the next name in this directory */
    }
    strcpy(fullpath,fpath);
    strcat(fullpath,"/");
    strcat(fullpath,dent->d_name);

  RESTART_STAT:
    if (stat(fullpath,&sbuf)) {
if (errno == EINTR) {
    if (--restart_cnt)
goto RESTART_STAT;
}
if (!(flags & FS_NOERROR))
    fprintf(stderr,"tgrep: Can’t stat file/dir %s, %s. "
    "Ignored.\n",
    fullpath,strerror(errno));
goto ERROR;
    }

    switch (sbuf.st_mode & S_IFMT) {
    case S_IFREG :
if (flags & TG_FILEPAT) {
    if (pmatch(pm_file_pat, dent->d_name, &pm_file_len)) {
DP(DLEVEL3,("file pat match (cascade) %s\n",
    dent->d_name));
add_work(fullpath,FILET);
    }
}
else {
    add_work(fullpath,FILET);
    DP(DLEVEL3,("cascade added file (MATCH) %s to Work Q\n",
fullpath));
}
break;
    case S_IFDIR :
DP(DLEVEL3,("cascade added dir %s to Work Q\n",fullpath));
add_work(fullpath,DIRT);
break;
    }
}

      ERROR:
closedir(dp);
      DONE:
free(wt->path);
free(wt);
notrun();
mutex_lock(&cascade_q_lk);
if (cascade_pool_cnt > cascade_thr_limit) {
    mutex_unlock(&cascade_q_lk);
    DP(DLEVEL5,("Cascade thread exiting\n"));
    if (flags & FS_STATS) {
mutex_lock(&stat_lk);
st_cascade_destroy++;
mutex_unlock(&stat_lk);
    }
    return(0); /* thr_exit */
}
else {
    DP(DLEVEL5,("Cascade thread waiting in pool\n"));
    cascade_pool_cnt++;
    while (!cascade_q)
cond_wait(&cascade_q_cv,&cascade_q_lk);
    cascade_pool_cnt--;
    wt = cascade_q;  /* we have work to do! */
    if (cascade_q->next)
cascade_q = cascade_q->next;
    else
cascade_q = NULL;
    mutex_unlock(&cascade_q_lk);
}
    }
    /*NOTREACHED*/
}

/*
 * Print Local Output: Called by the search thread after it is done searchi
 * a single file. If any oputput was saved (matching lines), the lines are
 * displayed as a group on stdout.
 */
int
print_local_output(out_t *out, work_t *wt)
{
    out_t       *pp, *op;
    int         out_count = 0;
    int         printed = 0;
    int  print_name = 1;

    pp = out;
    mutex_lock(&output_print_lk);
    if (pp && (flags & TG_PROGRESS)) {
        progress++;
        if (progress >= progress_offset) {
            progress = 0;
            fprintf(stderr,".");
        }
    }
    while (pp) {
out_count++;
if (!(flags & FC_COUNT)) {
    if (flags & FL_NAMEONLY) {  /* Pint name ONLY ! */
if (!printed) {
    printed = 1;
    printf("%s\n",wt->path);
}
    }
    else {  /* We are printing more then just the name */
if (!(flags & FH_HOLDNAME))  /* do not print name ? */
    printf("%s :",wt->path);
if (flags & FB_BLOCK)
    printf("%ld:",pp->byte_count/512+1);
if (flags & FN_NUMBER)
    printf("%d:",pp->line_count);
printf("%s\n",pp->line);
    }
}
        op = pp;
        pp = pp->next;
        /* free the nodes as we go down the list */
        free(op->line);
        free(op);
    }
    mutex_unlock(&output_print_lk);
    mutex_lock(&global_count_lk);
    global_count += out_count;
    mutex_unlock(&global_count_lk);
    return(0);
}

/*
 * add output local: is called by a search thread as it finds matching line
 * the matching line, it's byte offset, line count, etc are stored until th
 * search thread is done searching the file, then the lines are printed as
 * a group. This way the lines from more then a single file are not mixed
 * together.
 */
int
add_output_local(out_t **out, work_t *wt,int lc, long bc, char *line)
{
    out_t       *ot,*oo, *op;

    if (( ot = (out_t *)malloc(sizeof(out_t))) == NULL)
        goto ERROR;
    if (( ot->line = (char *)malloc(strlen(line)+1)) == NULL)
        goto ERROR;

    strcpy(ot->line,line);
    ot->line_count = lc;
    ot->byte_count = bc;

    if (!*out) {
        *out = ot;
        ot->next = NULL;
        return(0);
    }
    /* append to the END of the list, keep things sorted! */
    op = oo = *out;
    while(oo) {
        op = oo;
        oo = oo->next;
    }
    op->next = ot;
    ot->next = NULL;
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Output lost. No space. "
                "[%s: line %d byte %d match : %s\n",
                wt->path,lc,bc,line);
    return(1);
}

/*
 * print stats: If the -S flag is set, after ALL files have been searched,
 * main thread calls this function to print the stats it keeps on how the
 * search went.
 */
void
prnt_stats(void)
{
    float a,b,c;
    float t = 0.0;
    time_t  st_end = 0;
    char    tl[80];

    st_end = time(NULL); /* stop the clock */
    fprintf(stderr,"\n----------------- Tgrep Stats. --------------------\n");
    fprintf(stderr,"Number of directories searched:           %d\n",
    st_dir_search);
    fprintf(stderr,"Number of files searched:                 %d\n",
    st_file_search);
    c = (float)(st_dir_search + st_file_search) / (float)(st_end - st_start);
    fprintf(stderr,"Dir/files per second:                     %3.2f\n",
    c);
    fprintf(stderr,"Number of lines searched:                 %d\n",
    st_line_search);
    fprintf(stderr,"Number of matching lines to target:       %d\n",
    global_count);

    fprintf(stderr,"Number of cascade threads created:       %d\n",
    st_cascade);
    fprintf(stderr,"Number of cascade threads from pool:     %d\n",
    st_cascade_pool);
    a = st_cascade_pool; b = st_dir_search;
    fprintf(stderr,"Cascade thread pool hit rate:            %3.2f%%\n",
    ((a/b)*100));
    fprintf(stderr,"Cascade pool overall size:               %d\n",
    cascade_pool_cnt);
    fprintf(stderr,"Cascade pool size limit:                 %d\n",
    cascade_thr_limit);
    fprintf(stderr,"Number of cascade threads destroyed:     %d\n",
    st_cascade_destroy);

    fprintf(stderr,"Number of search threads created:        %d\n",
    st_search);
    fprintf(stderr,"Number of search threads from pool:      %d\n",
    st_pool);
    a = st_pool; b = st_file_search;
    fprintf(stderr,"Search thread pool hit rate:             %3.2f%%\n",
    ((a/b)*100));
    fprintf(stderr,"Search pool overall size:                %d\n",
    search_pool_cnt);
    fprintf(stderr,"Search pool size limit:                  %d\n",
    search_thr_limit);
    fprintf(stderr,"Number of search threads destroyed:      %d\n",
    st_destroy);

    fprintf(stderr,"Max # of threads running concurrenly:    %d\n",
    st_maxrun);
    fprintf(stderr,"Total run time, in seconds.              %d\n",
           (st_end - st_start));

    /* Why did we wait ? */
    a = st_workfds; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
```

```
    fprintf(stderr,"Work stopped due to no FD's:  (%.3d)       %d Times, %3.2f%%\n
           search_thr_limit,st_workfds,c);
    a = st_worknull; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    fprintf(stderr,"Work stopped due to no work on Q:         %d Times, %3.2f%%\n"
           st_worknull,c);
#ifndef __lock_lint  /* it is OK to read HERE with out the lock ! */
    if (tglimit == UNLIMITED)
        strcpy(tl,"Unlimited");
    else
        sprintf(tl,"   %.3d   ",tglimit);
#endif
    a = st_worklimit; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    fprintf(stderr,"Work stopped due to TGLIMIT:  (%.9s) %d Times, %3.2f%%\n",
           tl,st_worklimit,c);
    fprintf(stderr,"Work continued to be handed out:          %3.2f%%\n",
    100.00-t);
    fprintf(stderr,"----------------------------------------------------\n");
}

/*
 * not running: A glue function to track if any search threads or cascade
 * threads are running. When the count is zero, and the work Q is NULL,
 * we can safly say, WE ARE DONE.
 */
void
notrun (void)
{
    mutex_lock(&work_q_lk);
    work_cnt--;
    tglimit++;
    current_open_files++;
    mutex_lock(&running_lk);
    if (flags & FS_STATS) {
        mutex_lock(&stat_lk);
        if (running > st_maxrun) {
            st_maxrun = running;
    DP(DLEVEL6,("Max Running has increased to %d\n",st_maxrun));
}
        mutex_unlock(&stat_lk);
    }
    running--;
    if (work_cnt == 0 && running == 0) {
        all_done = 1;
        DP(DLEVEL6,("Setting ALL_DONE flag to TRUE.\n"));
    }
    mutex_unlock(&running_lk);
    cond_signal(&work_q_cv);
    mutex_unlock(&work_q_lk);
}

/*
 * uncase: A glue function. If the -i (case insensitive) flag is set, the
 * target strng and the read in line is converted to lower case before
 * comparing them.
 */
void
uncase(char *s)
{
    char        *p;

    for (p = s; *p != NULL; p++)
        *p = (char)tolower(*p);
}


/*
 * SigThread: if the -S option is set, the first ^C set to tgrep will
 * print the stats on the fly, the second will kill the process.
 */

void *
SigThread(void *arg)
{
    int sig;
    int stats_printed = 0;

    while (1) {
sig = sigwait(&set);
DP(DLEVEL7,("Signal %d caught\n",sig));
switch (sig) {
case -1:
    fprintf(stderr,"Signal error\n");
    break;
case SIGINT:
    if (stats_printed)
exit(1);
    stats_printed = 1;
    sig_print_stats();
    break;
case SIGHUP:
    sig_print_stats();
    break;
default:
    DP(DLEVEL7,("Default action taken (exit) for signal %d\n",sig));
    exit(1);  /* default action */
}
    }
}

void
sig_print_stats(void)
{
    /*
    ** Get the output lock first
    ** Then get the stat lock.
    */
    mutex_lock(&output_print_lk);
    mutex_lock(&stat_lk);
    prnt_stats();
    mutex_unlock(&stat_lk);
    mutex_unlock(&output_print_lk);
    return;
```

```
}

/*
 * usage: Have to have one of these.
 */
void
usage(void)
{
    fprintf(stderr,"usage: tgrep <options> pattern <{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Where:\n");
#ifdef DEBUG
    fprintf(stderr,"Debug     -d = debug level -d <levels> (-d0 for usage)\
    fprintf(stderr,"Debug     -f = block fd's from use (-f #)\n");
#endif
    fprintf(stderr,"          -b = show block count (512 byte block)\n");
    fprintf(stderr,"          -c = print only a line count\n");
    fprintf(stderr,"          -h = do not print file names\n");
    fprintf(stderr,"          -i = case insensitive\n");
    fprintf(stderr,"          -l = print file name only\n");
    fprintf(stderr,"          -n = print the line number with the line\n");
    fprintf(stderr,"          -s = Suppress error messages\n");
    fprintf(stderr,"          -v = print all but matching lines\n");
#ifdef NOT_IMP
    fprintf(stderr,"          -w = search for a \"word\"\n");
#endif
    fprintf(stderr,"          -r = Do not search for files in all "
                                "sub-directories\n");
    fprintf(stderr,"          -C = show continued lines (\"\\\")\n");
    fprintf(stderr,"          -p = File name regexp pattern. (Quote it)\n")
    fprintf(stderr,"          -P = show progress. -P 1 prints a DOT on stde
                   "               for each file it finds, -P 10 prints a D
                   "               on stderr for each 10 files it finds, et
    fprintf(stderr,"          -e = expression search.(regexp) More then one
    fprintf(stderr,"          -B = limit the number of threads to TGLIMIT\n
    fprintf(stderr,"          -S = Print thread stats when done.\n");
    fprintf(stderr,"          -Z = Print help on the regexp used.\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Notes:\n");
    fprintf(stderr,"      If you start tgrep with only a directory name\n");
    fprintf(stderr,"      and no file names, you must not have the -r option\n");
    fprintf(stderr,"      set or you will get no output.\n");
    fprintf(stderr,"      To search stdin (piped input), you must set -r\n");
    fprintf(stderr,"      Tgrep will search ALL files in ALL \n");
    fprintf(stderr,"      sub-directories. (like */* */*/* */*/*/* etc..)\n");
    fprintf(stderr,"      if you supply a directory name.\n");
    fprintf(stderr,"      If you do not supply a file, or directory name,\n");
    fprintf(stderr,"      and the -r option is not set, the current \n");
    fprintf(stderr,"      directory \".\" will be used.\n");
    fprintf(stderr,"      All the other options should work \"like\" grep\n");
    fprintf(stderr,"      The -p patten is regexp, tgrep will search only\n");
    fprintf(stderr,"      the file names that match the patten\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"      Tgrep Version %s\n",Tgrep_Version);
    fprintf(stderr,"\n");
    fprintf(stderr,"      Copy Right By Ron Winacott, 1993-1995.\n");
    fprintf(stderr,"\n");
    exit(0);
}

/*
 * regexp usage: Tell the world about tgrep custom (THREAD SAFE) regexp!
 */
int
regexp_usage (void)
{
    fprintf(stderr,"usage: tgrep <options> -e \"pattern\" <-e ...> "
    "<{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"metachars:\n");
    fprintf(stderr,"    . - match any character\n");
    fprintf(stderr,"    * - match 0 or more occurrences of pervious char\n");
    fprintf(stderr,"    + - match 1 or more occurrences of pervious char.\n");
    fprintf(stderr,"    ^ - match at begining of string\n");
    fprintf(stderr,"    $ - match end of string\n");
    fprintf(stderr,"    [ - start of character class\n");
    fprintf(stderr,"    ] - end of character class\n");
    fprintf(stderr,"    ( - start of a new pattern\n");
    fprintf(stderr,"    ) - end of a new pattern\n");
    fprintf(stderr,"    @(n)c - match <c> at column <n>\n");
    fprintf(stderr,"    | - match either pattern\n");
    fprintf(stderr,"    \\ - escape any special characters\n");
    fprintf(stderr,"    \\c - escape any special characters\n");
    fprintf(stderr,"    \\o - turn on any special characters\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"To match two diffrerent patterns in the same command\n"
    fprintf(stderr,"Use the or function. \n"
            "ie: tgrep -e \"(pat1)|(pat2)\" file\n"
            "This will match any line with \"pat1\" or \"pat2\" in it.\n");
    fprintf(stderr,"You can also use up to %d -e expresions\n",MAXREGEXP);
    fprintf(stderr,"RegExp Pattern matching brought to you by Marc Staveley
    exit(0);
}

/*
 * debug usage: If compiled with -DDEBUG, turn it on, and tell the world
 * how to get tgrep to print debug info on different threads.
 */
#ifdef DEBUG
void
debug_usage(void)
{
    int i = 0;

    fprintf(stderr,"DEBUG usage and levels:\n");
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"Level                   code\n");
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"0                 This message.\n");
    for (i=0; i<9; i++) {
        fprintf(stderr,"%d                   %s\n",i+1,debug_set[i].name);
    }
    fprintf(stderr,"--------------------------------------------------\n");
    fprintf(stderr,"You can or the levels together like -d134 for levels\n"
    fprintf(stderr,"1 and 3 and 4.\n");
    fprintf(stderr,"\n");
    exit(0);
}
#endif
```

31.13 Multithreaded Quicksort

The following example `tquick.c`implements the quicksort algorithm using threads.

```
/*
*  Multithreaded Demo Source
*
*  Copyright (C) 1995 by Sun Microsystems, Inc.
*  All rights reserved.
*
*  This file is a product of SunSoft, Inc. and is provided for
*  unrestricted use provided that this legend is included on all
*  media and as a part of the software program in whole or part.
*  Users may copy, modify or distribute this file at will.
*
*  THIS FILE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING
*  THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR
*  PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
*
*  This file is provided with no support and without any obligation on the
*  part of SunSoft, Inc. to assist in its use, correction, modification or
*  enhancement.
*
*  SUNSOFT AND SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT
*  TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS
*  FILE OR ANY PART THEREOF.
```

```
*
*  IN NO EVENT WILL SUNSOFT OR SUN MICROSYSTEMS, INC. BE LIABLE FOR ANY
*  LOST REVENUE OR PROFITS OR OTHER SPECIAL, INDIRECT AND CONSEQUENTIAL
*  DAMAGES, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH
*  DAMAGES.
*
*  SunSoft, Inc.
*  2550 Garcia Avenue
*  Mountain View, California  94043
*/

/*
 * multiple-thread quick-sort.  See man page for qsort(3c) for info.
 * Works fine on uniprocessor machines as well.
 *
 *  Written by:  Richard Pettit (Richard.Pettit@West.Sun.COM)
 */

#include <unistd.h>
#include <stdlib.h>
#include <thread.h>

/* don't create more threads for less than this */
#define SLICE_THRESH   4096

/* how many threads per lwp */
#define THR_PER_LWP       4

/* cast the void to a one byte quanitity and compute the offset */
#define SUB(a, n)      ((void *) (((unsigned char *) (a)) + ((n) * width)))

typedef struct {
  void    *sa_base;
  int      sa_nel;
  size_t   sa_width;
  int    (*sa_compar)(const void *, const void *);
} sort_args_t;
```

```
/* for all instances of quicksort */
static int threads_avail;

#define SWAP(a, i, j, width) \
{ \
  int n; \
  unsigned char uc; \
  unsigned short us; \
  unsigned long ul; \
  unsigned long long ull; \
 \
  if (SUB(a, i) == pivot) \
    pivot = SUB(a, j); \
  else if (SUB(a, j) == pivot) \
    pivot = SUB(a, i); \
 \
  /* one of the more convoluted swaps I've done */ \
  switch(width) { \
  case 1: \
    uc = *((unsigned char *) SUB(a, i)); \
    *((unsigned char *) SUB(a, i)) = *((unsigned char *) SUB(a, j)); \
    *((unsigned char *) SUB(a, j)) = uc; \
    break; \
  case 2: \
    us = *((unsigned short *) SUB(a, i)); \
    *((unsigned short *) SUB(a, i)) = *((unsigned short *) SUB(a, j)); \
    *((unsigned short *) SUB(a, j)) = us; \
    break; \
  case 4: \
    ul = *((unsigned long *) SUB(a, i)); \
    *((unsigned long *) SUB(a, i)) = *((unsigned long *) SUB(a, j)); \
    *((unsigned long *) SUB(a, j)) = ul; \
    break; \
  case 8: \
    ull = *((unsigned long long *) SUB(a, i)); \
    *((unsigned long long *) SUB(a,i)) = *((unsigned long long *) SUB(a,j)); \
    *((unsigned long long *) SUB(a, j)) = ull; \
    break; \
  default: \
    for(n=0; n<width; n++) { \
      uc = ((unsigned char *) SUB(a, i))[n]; \
      ((unsigned char *) SUB(a, i))[n] = ((unsigned char *) SUB(a, j))[n];
      ((unsigned char *) SUB(a, j))[n] = uc; \
    } \
    break; \
  } \
}

static void *
_quicksort(void *arg)
{
  sort_args_t *sargs = (sort_args_t *) arg;
  register void *a = sargs->sa_base;
  int n = sargs->sa_nel;
  int width = sargs->sa_width;
  int (*compar)(const void *, const void *) = sargs->sa_compar;
  register int i;
  register int j;
  int z;
  int thread_count = 0;
  void *t;
  void *b[3];
  void *pivot = 0;
  sort_args_t sort_args[2];
  thread_t tid;

  /* find the pivot point */
  switch(n) {
  case 0:
  case 1:
    return 0;
  case 2:
    if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
      SWAP(a, 0, 1, width);
    }
    return 0;
```

```
case 3:
  /* three sort */
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, 2), SUB(a, 1)) < 0) {
    SWAP(a, 2, 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, 1), SUB(a, 0)) < 0) {
    SWAP(a, 1, 0, width);
  }
  return 0;
default:
  if (n > 3) {
    b[0] = SUB(a, 0);
    b[1] = SUB(a, n / 2);
    b[2] = SUB(a, n - 1);
    /* three sort */
    if ((*compar)(b[0], b[1]) > 0) {
      t = b[0];
      b[0] = b[1];
      b[1] = t;
    }
    /* the first two are now ordered, now order the second two */
    if ((*compar)(b[2], b[1]) < 0) {
      t = b[1];
      b[1] = b[2];
      b[2] = t;
    }
    /* should the second be moved to the first? */
    if ((*compar)(b[1], b[0]) < 0) {
      t = b[0];
      b[0] = b[1];
      b[1] = t;
    }
    if ((*compar)(b[0], b[2]) != 0)
```

```
      if ((*compar)(b[0], b[1]) < 0)
        pivot = b[1];
      else
        pivot = b[2];
  }
  break;
}
if (pivot == 0)
  for(i=1; i<n; i++) {
    if (z = (*compar)(SUB(a, 0), SUB(a, i))) {
      pivot = (z > 0) ? SUB(a, 0) : SUB(a, i);
      break;
    }
  }
if (pivot == 0)
  return;

/* sort */
i = 0;
j = n - 1;
while(i <= j) {
  while((*compar)(SUB(a, i), pivot) < 0)
    ++i;
  while((*compar)(SUB(a, j), pivot) >= 0)
    --j;
  if (i < j) {
    SWAP(a, i, j, width);
    ++i;
    --j;
  }
}

/* sort the sides judiciously */
switch(i) {
case 0:
case 1:
  break;
case 2:
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  break;
case 3:
  /* three sort */
  if ((*compar)(SUB(a, 0), SUB(a, 1)) > 0) {
    SWAP(a, 0, 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, 2), SUB(a, 1)) < 0) {
    SWAP(a, 2, 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, 1), SUB(a, 0)) < 0) {
    SWAP(a, 1, 0, width);
  }
  break;
default:
  sort_args[0].sa_base          = a;
  sort_args[0].sa_nel           = i;
  sort_args[0].sa_width         = width;
  sort_args[0].sa_compar        = compar;
  if ((threads_avail > 0) && (i > SLICE_THRESH)) {
    threads_avail--;
    thr_create(0, 0, _quicksort, &sort_args[0], 0, &tid);
    thread_count = 1;
  } else
    _quicksort(&sort_args[0]);
  break;
}
j = n - i;
switch(j) {
case 1:
  break;
case 2:
  if ((*compar)(SUB(a, i), SUB(a, i + 1)) > 0) {
    SWAP(a, i, i + 1, width);
  }
  break;
case 3:
  /* three sort */
  if ((*compar)(SUB(a, i), SUB(a, i + 1)) > 0) {
    SWAP(a, i, i + 1, width);
  }
  /* the first two are now ordered, now order the second two */
  if ((*compar)(SUB(a, i + 2), SUB(a, i + 1)) < 0) {
    SWAP(a, i + 2, i + 1, width);
  }
  /* should the second be moved to the first? */
  if ((*compar)(SUB(a, i + 1), SUB(a, i)) < 0) {
    SWAP(a, i + 1, i, width);
  }
  break;
default:
  sort_args[1].sa_base          = SUB(a, i);
  sort_args[1].sa_nel           = j;
  sort_args[1].sa_width         = width;
  sort_args[1].sa_compar        = compar;
  if ((thread_count == 0) && (threads_avail > 0) && (i > SLICE_THRESH)) {
    threads_avail--;
    thr_create(0, 0, _quicksort, &sort_args[1], 0, &tid);
    thread_count = 1;
  } else
    _quicksort(&sort_args[1]);
  break;
}
if (thread_count) {
  thr_join(tid, 0, 0);
  threads_avail++;
}
return 0;
}

void
quicksort(void *a, size_t n, size_t width,
      int (*compar)(const void *, const void *))
{
static int ncpus = -1;
sort_args_t sort_args;

if (ncpus == -1) {
  ncpus = sysconf( _SC_NPROCESSORS_ONLN);

  /* lwp for each cpu */
  if ((ncpus > 1) && (thr_getconcurrency() < ncpus))
    thr_setconcurrency(ncpus);

  /* thread count not to exceed THR_PER_LWP per lwp */
  threads_avail = (ncpus == 1) ? 0 : (ncpus * THR_PER_LWP);
}
sort_args.sa_base = a;
sort_args.sa_nel = n;
sort_args.sa_width = width;
sort_args.sa_compar = compar;
(void) _quicksort(&sort_args);
}
```

Chapter 32

Remote Procedure Calls (RPC)

This chapter provides an overview of Remote Procedure Calls (RPC) RPC.

32.1 What Is RPC

RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

RPC makes the client/server model of computing more powerful and easier to program. When combined with the ONC RPCGEN protocol compiler (Chapter 33) clients transparently make remote calls through a local procedure interface.

32.2 How RPC Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure 32.1

shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

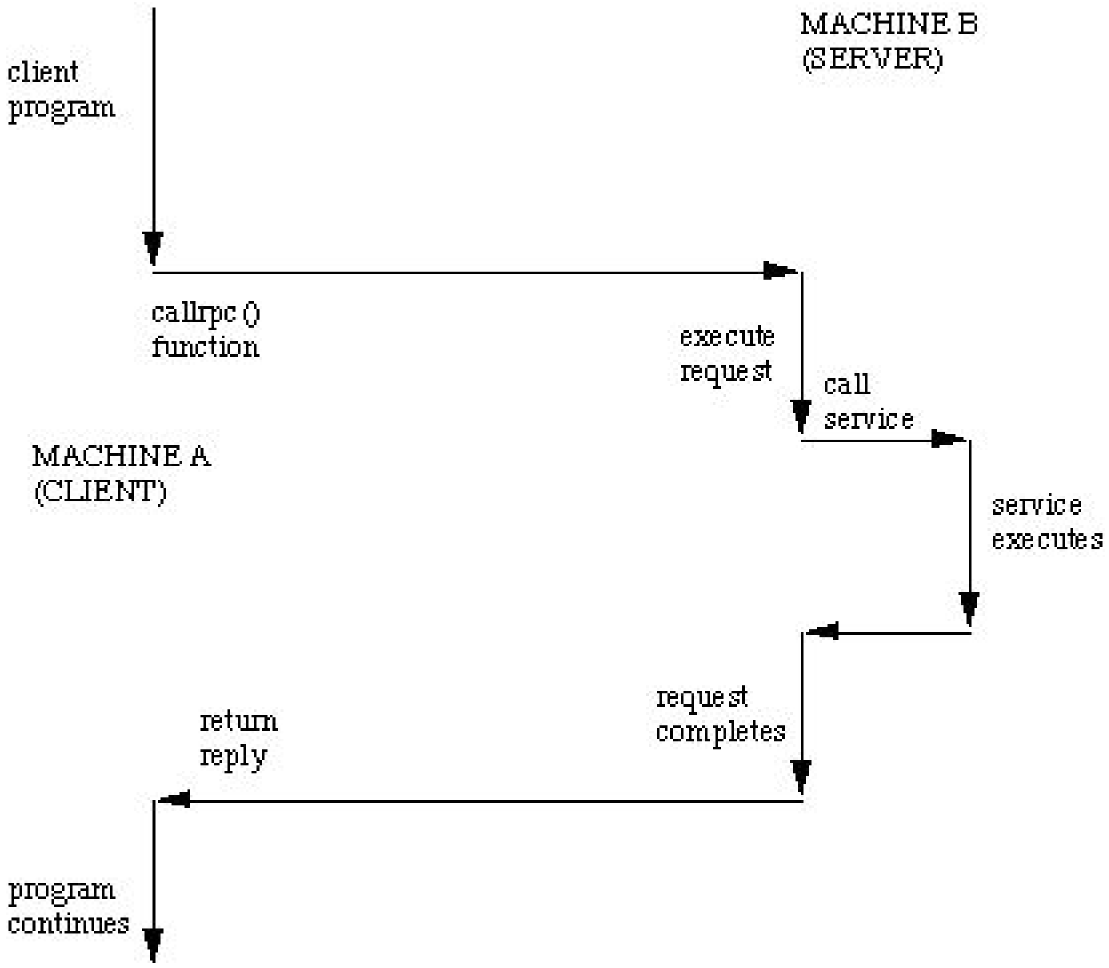


Figure 32.1: Remote Procedure Calling Mechanism

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) The program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously. Each version contains a a number of procedures that can be called remotely. Each procedure has a procedure number.

32.3 RPC Application Development

Consider an example:

A client/server lookup in a personal database on a remote machine. Assuming that we cannot access the database from the local machine (via NFS).

We use UNIX to run a remote shell and execute the command this way. There are some problems with this method:

- the command may be slow to execute.
- You require an login account on the remote machine.

The RPC alternative is to

- establish an server on the remote machine that can repond to queries.
- Retrieve information by calling a query which will be quicker than previous approach.

To develop an RPC application the following steps are needed:

- Specify the protocol for client server communication
- Develop the client program
- Develop the server program

The programs will be compiled seperately. The communication protocol is achieved by generated stubs and these stubs and rpc (and other libraries) will need to be linked in.

32.3.1 Defining the Protocol

The easiest way to define and generate the protocol is to use a protocol complier such as `rpcgen` which we discuss is Chapter 33.

For the protocol you must identify the name of the service procedures, and data types of parameters and return arguments.

The protocol compiler reads a definitio and automatically generates client and server stubs.

`rpcgen` uses its own language (RPC language or RPCL) which looks very similar to preprocessor directives.

`rpcgen` exists as a standalone executable compiler that reads special files denoted by a `.x` prefix.

So to compile a RPCL file you simply do

```
rpcgen rpcprog.x
```

This will generate possibly four files:

- `rpcprog_clnt.c` — the client stub
- `rpcprog_svc.c` — the server stub
- `rpcprog_xdr.c` — If necessary XDR (external data representation) filters
- `rpcprog.h` — the header file needed for any XDR filters.

The external data representation (XDR) is an data abstraction needed for machine independent communication. The client and server need not be machines of the same type.

32.3.2 Defining Client and Server Application Code

We must now write the the client and application code. They must communicate via procedures and data types specified in the Protocol.

The service side will have to register the procedures that may be called by the client and receive and return any data required for processing.

The client application call the remote procedure pass any required data and will receive the retruned data.

There are several levels of application interfaces that may be used to develop RPC applications. We will briefly disuss these below before exapnading thhe most common of these in later chapters.

32.3.3 Compliling and running the application

Let us consider the full compilation model required to run a RPC application. Makefiles are useful for easing the burden of compiling RPC applications but it is necessary to understand the complete model before one can assemble a convenient makefile.

Assume the the client program is called `rpcprog.c`, the service program is `rpcsvc.c` and that the protocol has been defined in `rpcprog.x` and that `rpcgen` has been used to produce the stub and filter files: `rpcprog_clnt.c, rpcprog_svc.c, rpcprog_xdr.c, rpcprog.h`.

The client and server program must include (`#include "rpcprog.h"`
You must then:

- compile the client code:

```
cc -c  rpcprog.c
```

- compile the client stub:

```
cc -c  rpcprog_clnt.c
```

- compile the XDR filter:

```
cc -c  rpcprog_xdr.c
```

- build the client executable:

```
cc -o rpcprog rpcprog.o rpcprog_clnt.o rpcprog_xdr.c
```

- compile the service procedures:

```
cc -c  rpcsvc.c
```

- compile the server stub:

```
cc -c  rpcprog_svc.c
```

- build the server executable:

```
cc -o rpcsvc rpcsvc.o rpcprog_svc.o rpcprog_xdr.c
```

Now simply run the programs `rpcprog` and `rpcsvc` on the client and server respectively. The server procedures must be registered before the client can call them.

32.4 Overview of Interface Routines

RPC has multiple levels of application interface to its services. These levels provide different degrees of control balanced with different amounts of interface code to implement. In order of increasing control and complexity. This section gives a summary of the routines available at each level. Simplified Interface Routines

The simplified interfaces are used to make remote procedure calls to routines on other machines, and specify only the type of transport to use. The routines at this level are used for most applications. Descriptions and code samples can be found in the section, Simplified Interface @ 3-2.

32.4.1 Simplified Level Routine Function

`rpc_reg()` — Registers a procedure as an RPC program on all transports of the specified type.

`rpc_call()` — Remote calls the specified procedure on the specified remote host.

`rpc_broadcast()` — Broadcasts a call message across all transports of the specified type. Standard Interface Routines The standard interfaces are divided into top level, intermediate level, expert level, and bottom level. These interfaces give a developer much greater control over communication parameters such as the transport being used, how long to wait beforeresponding to errors and retransmitting requests, and so on.

32.4.2 Top Level Routines

At the top level, the interface is still simple, but the program has to create a client handle before making a call or create a server handle before receiving calls. If you want the application to run on all transports, use this interface. Use of these routines and code samples can be found in Top Level Interface

`clnt_create()` — Generic client creation. The program tells `clnt_create()` where the server is located and the type of transport to use.

`clnt_create_timed()` Similar to `clnt_create()` but lets the programmer specify the maximum time allowed for each type of transport tried during the creation attempt.

`svc_create()` — Creates server handles for all transports of the specified type. The program tells `svc_create()` which dispatch function to use.

`clnt_call()` — Client calls a procedure to send a request to the server.

32.5 Intermediate Level Routines

The intermediate level interface of RPC lets you control details. Programs written at these lower levels are more complicated but run more efficiently. The intermediate level enables you to specify the transport to use.

`clnt_tp_create()` — Creates a client handle for the specified transport.

`clnt_tp_create_timed()` — Similar to `clnt_tp_create()` but lets the programmer specify the maximum time allowed. `svc_tp_create()` Creates a server handle for the specified transport.

`clnt_call()` — Client calls a procedure to send a request to the server.

32.5.1 Expert Level Routines

The expert level contains a larger set of routines with which to specify transport-related parameters. Use of these routines

`clnt_tli_create()` — Creates a client handle for the specified transport.

`svc_tli_create()` — Creates a server handle for the specified transport.

`rpcb_set()` — Calls rpcbind to set a map between an RPC service and a network address.

`rpcb_unset()` — Deletes a mapping set by `rpcb_set()`.

`rpcb_getaddr()` — Calls rpcbind to get the transport addresses of specified RPC services.

`svc_reg()` — Associates the specified program and version number pair with the specified dispatch routine.

`svc_unreg()` —- Deletes an association set by `svc_reg()`.

`clnt_call()` — Client calls a procedure to send a request to the server.

32.5.2 Bottom Level Routines

The bottom level contains routines used for full control of transport options.

`clnt_dg_create()` — Creates an RPC client handle for the specified remote program, using a connectionless transport.

`svc_dg_create()` — Creates an RPC server handle, using a connectionless transport.

`clnt_vc_create()` — Creates an RPC client handle for the specified remote program, using a connection-oriented transport.

`svc_vc_create()` — Creates an RPC server handle, using a connection-oriented transport.

`clnt_call()` — Client calls a procedure to send a request to the server.

32.6 The Programmer's Interface to RPC

This section addresses the C interface to RPC and describes how to write network applications using RPC. For a complete specification of the routines in the RPC library, see the `rpc` and related `man` pages.

32.6.1 Simplified Interface

The simplified interface is the easiest level to use because it does not require the use of any other RPC routines. It also limits control of the underlying communications mechanisms. Program development at this level can be rapid, and is directly supported by the `rpcgen` compiler. For most applications, rpcgen and its facilities are sufficient. Some RPC services are not available as C functions, but they are available as RPC programs. The simplified interface library routines provide direct access to the RPC facilities for programs that do not require fine levels of control.

Routines such as `rusers` are in the RPC services library `librpcsvc`. `rusers.c`, below, is a program that displays the number of users on a remote host. It calls the RPC library routine, `rusers`.

The `program.c` program listing:

```
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <stdio.h>

/*
* a program that calls the
* rusers() service
*/

main(int argc,char **argv)
```

```
{
int num;
if (argc != 2) {
   fprintf(stderr, "usage: %s hostname\n",
   argv[0]);
   exit(1);
   }

if ((num = rnusers(argv[1])) < 0) {
   fprintf(stderr, "error: rusers\n");
   exit(1);
  }

fprintf(stderr, "%d users on %s\n", num, argv[1] );
exit(0);
}
```

Compile the program with:

```
cc program.c -lrpcsvc -lnsl
```

The Client Side

There is just one function on the client side of the simplified interface `rpc_call()`.

It has nine parameters:

```
int
rpc_call (char *host /* Name of server host */,
    u_long prognum /* Server program number */,
    u_long versnum /* Server version number */,
    xdrproc_t inproc /* XDR filter to encode arg */,
    char *in /* Pointer to argument */,
    xdr_proc_t outproc /* Filter to decode result */,
    char *out /* Address to store result */,
    char *nettype /* For transport selection */);
```

This function calls the procedure specified by `prognum`, `versum`, and `procnum` on the host. The argument to be passed to the remote procedure is pointed to by the `in` parameter, and `inproc` is the XDR filter to encode this argument. The `out` parameter is an address where the result from the remote procedure is to be placed. `outproc` is an XDR filter which will decode the result and place it at this address.

The client blocks on `rpc_call()` until it receives a reply from the server. If the server accepts, it returns `RPC_SUCCESS` with the value of zero. It will return a non-zero value if the call was unsuccessful. This value can be cast to the type `clnt_stat`, an enumerated type defined in the RPC include files (<rpc/rpc.h>) and interpreted by the `clnt_sperrno()` function. This function returns a pointer to a standard RPC error message corresponding to the error code. In the example, all "visible" transports listed in `/etc/netconfig` are tried. Adjusting the number of retries requires use of the lower levels of the RPC library. Multiple arguments and results are handled by collecting them in structures.

The example changed to use the simplified interface, looks like

```
#include <stdio.h>
#include <utmp.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

/* a program that calls the RUSERSPROG
* RPC program
*/

main(int argc, char **argv)

{
   unsigned long nusers;
   enum clnt_stat cs;
   if (argc != 2) {
     fprintf(stderr, "usage: rusers hostname\n");
     exit(1);
    }

   if( cs = rpc_call(argv[1], RUSERSPROG,
          RUSERSVERS, RUSERSPROC_NUM, xdr_void,
          (char *)0, xdr_u_long, (char *)&nusers,
          "visible") != RPC_SUCCESS ) {
              clnt_perrno(cs);
              exit(1);
           }

   fprintf(stderr, "%d users on %s\n", nusers, argv[1] );
   exit(0);
}
```

Since data types may be represented differently on different machines, `rpc_call()` needs both the type of, and a pointer to, the RPC argument (similarly for the result). For `RUSERSPROC_NUM`, the return value is an unsigned long, so the first return parameter of `rpc_call()` is `xdr_u_long` (which is for an unsigned long) and the second is `&nusers` (which points to unsigned long storage). Because `RUSERSPROC_NUM` has no argument, the XDR encoding function of `rpc_call()` is `xdr_void()` and its argument is `NULL`.

The Server Side

The server program using the simplified interface is very straightforward. It simply calls `rpc_reg()` to register the procedure to be called, and then it calls `svc_run()`, the RPC library's remote procedure dispatcher, to wait for requests to come in.

`rpc_reg()` has the following prototype:

```
rpc_reg(u_long prognum /* Server program number */,
        u_long versnum /* Server version number */,
        u_long procnum /* server procedure number */,
        char *procname /* Name of remote function */,
        xdrproc_t inproc /* Filter to encode arg */,
        xdrproc_t outproc /* Filter to decode result */,
        char *nettype /* For transport selection */);
```

`svc_run()` invokes service procedures in response to RPC call messages. The dispatcher in `rpc_reg()` takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered. Some notes about the server program:

- Most RPC applications follow the naming convention of appending a `_1` to the function name. The sequence `_n` is added to the procedure names to indicate the version number `n` of the service.

- The argument and result are passed as addresses. This is true for all functions that are called remotely. If you pass `NULL` as a result of a function, then no reply is sent to the client. It is assumed that there is no reply to send.
- The result must exist in static data space because its value is accessed after the actual procedure has exited. The RPC library function that builds the RPC reply message accesses the result and sends the value back to the client.
- Only a single argument is allowed. If there are multiple elements of data, they should be wrapped inside a structure which can then be passed as a single entity.
- The procedure is registered for each transport of the specified type. If the type parameter is `(char *)NULL`, the procedure is registered for all transports specified in `NETPATH`.

You can sometimes implement faster or more compact code than can `rpcgen`. `rpcgen` handles the generic code-generation cases. The following program is an example of a hand-coded registration routine. It registers a single procedure and enters `svc_run()` to service requests.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

void *rusers();

main()
{
  if(rpc_reg(RUSERSPROG, RUSERSVERS,
        RUSERSPROC_NUM, rusers,
        xdr_void, xdr_u_long,
        "visible") == -1) {
           fprintf(stderr, "Couldn't Register\n");
            exit(1);
          }
   svc_run(); /* Never returns */
  fprintf(stderr, "Error: svc_run returned!\n");
  exit(1);
}
```

rpc_reg() can be called as many times as is needed to register different programs, versions, and procedures.

32.6.2 Passing Arbitrary Data Types

Data types passed to and received from remote procedures can be any of a set of predefined types, or can be programmer-defined types. RPC handles arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a standard transfer format called external data representation (XDR) before sending them over the transport. The conversion from a machine representation to XDR is called serializing, and the reverse process is called deserializing. The translator arguments of rpc_call() and rpc_reg() can specify an XDR primitive procedure, like xdr_u_long(), or a programmer-supplied routine that processes a complete argument structure. Argument processing routines must take only two arguments: a pointer to the result and a pointer to the XDR handle.

The following XDR Primitive Routines are available:

```
xdr_int() xdr_netobj() xdr_u_long() xdr_enum()
xdr_long() xdr_float() xdr_u_int() xdr_bool()
xdr_short() xdr_double() xdr_u_short() xdr_wrapstring()
xdr_char() xdr_quadruple() xdr_u_char() xdr_void()
```

The nonprimitive xdr_string(), which takes more than two parameters, is called from xdr_wrapstring().

For an example of a programmer-supplied routine, the structure:

```
struct simple {
   int a;
   short b;
  } simple;
```

contains the calling arguments of a procedure. The XDR routine xdr_simple() translates the argument structure as shown below:

```
#include <rpc/rpc.h>
#include "simple.h"

bool_t xdr_simple(XDR *xdrsp, struct simple *simplep)

{
   if (!xdr_int(xdrsp, &simplep->a))
      return (FALSE);
   if (!xdr_short(xdrsp, &simplep->b))
      return (FALSE);
   return (TRUE);
}
```

An equivalent routine can be generated automatically by `rpcgen` (See Chapter 33).

An XDR routine returns nonzero (a C TRUE) if it completes successfully, and zero otherwise.

For more complex data structures use the XDR prefabricated routines:

```
xdr_array() xdr_bytes() xdr_reference()
xdr_vector() xdr_union() xdr_pointer()
xdr_string() xdr_opaque()
```

For example, to send a variable-sized array of integers, it is packaged in a structure containing the array and its length:

```
struct varintarr {
int *data;
int arrlnth;
} arr;
```

Translate the array with `xdr_array()`, as shown below:

```
bool_t xdr_varintarr(XDR *xdrsp, struct varintarr *arrp)

{
    return(xdr_array(xdrsp, (caddr_t)&arrp->data,
              (u_int *)&arrp->arrlnth, MAXLEN, sizeof(int), xdr_int));
}
```

```
\end{vebatim}

The arguments of {\tt xdr\_array()} are the XDR handle, a pointer to the array,
 a pointer to
the size of the array, the maximum array size, the size of each array element, and
pointer to the XDR routine to translate each array element.

If the size of the array is
known in advance, use {\tt xdr\_vector()} instread as is more efficient:

\begin{verbatim}

int intarr[SIZE];

bool_t xdr_intarr(XDR *xdrsp, int intarr[])
{
   return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int), xdr_int));
}
```

XDR converts quantities to 4-byte multiples when serializing. For arrays of characters, each character occupies 32 bits. `xdr_bytes()` packs characters. It has four parameters similar to the first four parameters of `xdr_array()`.

Null-terminated strings are translated by `xdr_string()`. It is like `xdr_bytes()` with no length parameter. On serializing it gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

`xdr_reference()` calls the built-in functions $\hat{x}$dr_string() and `xdr_reference()`, which translates pointers to pass a string, and struct simple from the previous examples. An example use of `xdr_reference()` is as follows:

```
struct finalexample {
    char *string;
    struct simple *simplep;
   } finalexample;

bool_t xdr_finalexample(XDR *xdrsp, struct finalexample *finalp)

{  if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (FALSE);
   if (!xdr_reference( xdrsp, &finalp->simplep, sizeof(struct simple), xdr_
       return (FALSE);
    return (TRUE);
}
```

Note thatxdr_simple() could have been called here instead of xdr_reference()
.

32.6.3 Developing High Level RPC Applications

Let us now introduce some further functions and see how we develop an application using high level RPC routines. We will do this by studying an example.

We will develop a remote directory reading utility.

Let us first consider how we would write a local directory reader. We have seem how to do this already in Chapter 19.

Consider the program to consist of two files:

- `lls.c` — the main program which calls a routine in a local module `read_dir.c`

```
/*
 * ls.c: local directory listing main - before RPC
 */
#include <stdio.h>
#include <strings.h>
#include "rls.h"

main (int argc, char **argv)

{
        char    dir[DIR_SIZE];

        /* call the local procedure */
        strcpy(dir, argv[1]); /* char dir[DIR_SIZE] is coming and goin
        read_dir(dir);
        /* spew-out the results and bail out of here! */
        printf("%s\n", dir);

        exit(0);
}
```

- read_dir.c — the file containing the *local* routine read_dir().

```
/* note - RPC compliant procedure calls take one input and
   return one output. Everything is passed by pointer.  Return
   values should point to static data, as it might have to
   survive some while. */
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>     /* use <xpg2include/sys/dirent.h> (SunOS4.1) or
        <sys/dirent.h> for X/Open Portability Guide, issue 2 conformance */
#include "rls.h"

read_dir(char    *dir)
   /* char dir[DIR_SIZE] */
{
        DIR * dirp;
        struct direct *d;
      printf("beginning ");

        /* open directory */
        dirp = opendir(dir);
        if (dirp == NULL)
                return(NULL);

        /* stuff filenames into dir buffer */
        dir[0] = NULL;
        while (d = readdir(dirp))
                sprintf(dir, "%s%s\n", dir, d->d_name);

        /* return the result */
  printf("returning ");
```

```
          closedir(dirp);
          return((int)dir);  /* this is the only new line from Example 4
}
```

- the header file `rls.h` contains only the following (for now at least)

  ```
  #define DIR_SIZE 8192
  ```

  Clearly we need to share the size between the files. Later when we develop RPC versions more information will need to be added to this file.

This local program would be compiled as follows:

```
cc lls.c read_dir.c -o lls
```

Now we want to modify this program to work over a network: Allowing us to inspect directories of a remote server accross a network.

The following steps will be required:

- We will have to convert the `read_dir.c`, to run on the server.
  - We will have to register the server and the routine `read_dir()` on the server/.
- The client `lls.c` will have to call the routine as a remote procedure.
- We will have to define the protocol for communication between the client and the server programs.

Defining the protocol

We can can use simple `NULL`-terminated strings for passing and receivong the directory name and directory contents. Furthermore, we can embed the passing of these parameters directly in the client and server code.

We therefore need to specify the program, procedure and version numbers for client and servers. This can be done automatically using `rpcgen` or relying on prdefined macros in the simlified interface. Here we will specify them manually.

The server and client must agree *ahead of time* what logical adresses thney will use (The physical addresses do not matter they are hidden from the application developer)

Program numbers are defined in a standard way:

- $0x00000000$ – $0x1FFFFFFF$: Defined by Sun
- $0x20000000$ – $0x3FFFFFFF$: User Defined
- $0x40000000$ – $0x5FFFFFFF$: Transient
- $0x60000000$ – $0xFFFFFFFF$: Reserved

We will simply choose a *user deifnined value* for our program number. The version and procedure numbers are set according to standard practice.

We still have the `DIR_SIZE` definition required from the local version as the size of the directory buffer is rewquired by bith client and server programs.

Our new `rls.h` file contains:

```
#define DIR_SIZE 8192
#define DIRPROG ((u_long) 0x20000001)   /* server program (suite) number */
#define DIRVERS ((u_long) 1)    /* program version number */
#define READDIR ((u_long) 1)    /* procedure number for look-up */
```

32.6.4 Sharing the data

We have mentioned previously that we can pass the data a simple strings. We need to define an XDR filter routine `xdr_dir()` that shares the data. Recall that only one encoding and decoding argument can be handled. This is easy and defined via the standard `xdr_string()` routine.

The XDR file, `rls_xrd.c`, is as follows:

```
#include <rpc/rpc.h>

#include "rls.h"

bool_t xdr_dir(XDR *xdrs, char *objp)

{ return ( xdr_string(xdrs, &objp, DIR_SIZE) ); }
```

The Server Side

We can use the original `read_dir.c` file. All we need to do is register the procedure and start the server.

The procedure is registered with `registerrpc()` function. This is prototypes by:

```
registerrpc(u_long prognum /* Server program number */,
        u_long versnum /* Server version number */,
        u_long procnum /* server procedure number */,
        char *procname /* Name of remote function */,
        xdrproc_t inproc /* Filter to encode arg */,
        xdrproc_t outproc /* Filter to decode result */);
```

The parameters a similarly defined as in the `rpc_reg` simplified interface function. We have already discussed the setting of the parametere with the protocol `rls.h` header files and the `rls_xrd.c` XDR filter file.

The `svc_run()` routine has also been discussed previously.

The full `rls_svc.c` code is as follows:

```
#include <rpc/rpc.h>
#include "rls.h"

main()
{
        extern bool_t xdr_dir();
        extern char * read_dir();

        registerrpc(DIRPROG, DIRVERS, READDIR,
                        read_dir, xdr_dir, xdr_dir);

        svc_run();
}
```

The Client Side

At the client side we simply need to call the remote procedure. The function `callrpc()` does this. It is prototyped as follows:

```
callrpc(char *host /* Name of server host */,
    u_long prognum /* Server program number */,
    u_long versnum /* Server version number */,
    char *in /* Pointer to argument */,
    xdrproc_t inproc /* XDR filter to encode arg */,
    char *out /* Address to store result */
    xdr_proc_t outproc /* Filter to decode result */);
```

We call a local function `read_dir()` which uses `callrpc()` to call the remote procedure that has been registered `READDIR` at the server.

The full `rls.c` program is as follows:

```
/*
 * rls.c: remote directory listing client
 */
#include <stdio.h>
#include <strings.h>
#include <rpc/rpc.h>
#include "rls.h"

main (argc, argv)
int argc; char *argv[];
{
char    dir[DIR_SIZE];

        /* call the remote procedure if registered */
        strcpy(dir, argv[2]);
        read_dir(argv[1], dir); /* read_dir(host, directory) */

        /* spew-out the results and bail out of here! */
        printf("%s\n", dir);

        exit(0);
}

read_dir(host, dir)
char   *dir, *host;
{
        extern bool_t xdr_dir();
        enum clnt_stat clnt_stat;

        clnt_stat = callrpc ( host, DIRPROG, DIRVERS, READDIR,
                        xdr_dir, dir, xdr_dir, dir);
        if (clnt_stat != 0) clnt_perrno (clnt_stat);
}
```

32.7 Exercise

Exercise 32.1 *Compile and run the remote directory example* `rls.c` etc. *Run both the client and server locally and if possible over a network.*

Chapter 33

Protocol Compiling and Lower Level RPC Programming

This chapter introduces the rpcgen tool and provides a tutorial with code examples and usage of the available compile-time flags. We also introduce some further RPC programming routines.

33.1 What is `rpcgen`

The `rpcgen` tool generates remote program interface modules. It compiles source code written in the RPC Language. RPC Language is similar in syntax and structure to C. `rpcgen` produces one or more C language source modules, which are then compiled by a C compiler.

The default output of rpcgen is:

- A header file of definitions common to the server and the client
- A set of XDR routines that translate each data type defined in the header file
- A stub program for the server
- A stub program for the client

`rpcgen` can optionally generate (although we *do not* consider these issues here — see `man` pages or receommended reading):

- Various transports
- A time-out for servers
- Server stubs that are MT safe
- Server stubs that are not main programs
- C-style arguments passing ANSI C-compliant code
- An RPC dispatch table that checks authorizations and invokes service routines

`rpcgen` significantly reduces the development time that would otherwise be spent developing low-level routines. Handwritten routines link easily with the rpcgen output.

33.2 An `rpcgen` Tutorial

`rpcgen` provides programmers a simple and direct way to write distributed applications. Server procedures may be written in any language that observes procedure-calling conventions. They are linked with the server stub produced by rpcgen to form an executable server program. Client procedures are written and linked in the same way. This section presents some basic rpcgen programming examples. Refer also to the `man rpcgen` online manual page.

33.2.1 Converting Local Procedures to Remote Procedures

Assume that an application runs on a single computer and you want to convert it to run in a "distributed" manner on a network. This example shows the stepwise conversion of this program that writes a message to the system console.

Single Process Version of `printmesg.c`:

```
/* printmsg.c: print a message on the console */
#include <stdio.h>
main(int argc, char *argv[])
```

```
{
   char *message;
   if (argc != 2) {
      fprintf(stderr, "usage: %s <message>\n",argv[0]);
      exit(1);
    }
   message = argv[1];
   if (!printmessage(message)) {
     fprintf(stderr,"%s: couldnt print your message\n",argv[0]);
     exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}

/* Print a message to the console.
* Return a boolean indicating whether
* the message was actually printed. */

printmessage(char *msg)

{
   FILE *f;
   f = fopen("/dev/console", "w");
   if (f == (FILE *)NULL) {
     return (0);
    }
   fprintf(f, "%s\n", msg);
   fclose(f);
   return(1);
}
```

For local use on a single machine, this program could be compiled and executed as follows:

```
$ cc printmsg.c -o printmsg
$ printmsg "Hello, there."
Message delivered!
$
```

If the `printmessage()` function is turned into a *remote procedure*, it can be called from anywhere in the network. `rpcgen` makes it easy to do this:

First, determine the data types of all procedure-calling arguments and the result argument. The calling argument of `printmessage()` is a string, and the result is an integer. We can write a protocol specification in RPC language that describes the remote version of printmessage. The RPC language source code for such a specification is:

```
/* msg.x: Remote msg printing protocol */
program MESSAGEPROG {
   version PRINTMESSAGEVERS {
     int PRINTMESSAGE(string) = 1;
   } = 1;
} = 0x20000001;
```

Remote procedures are always declared as part of remote programs. The code above declares an entire remote program that contains the single procedure `PRINTMESSAGE`.

In this example,

- `PRINTMESSAGE` procedure is declared to be:
  - the `procedure 1`,
  - in version 1 of the remote program
- `MESSAGEPROG,` with the program number 0x20000001.

Version numbers are incremented when functionality is changed in the remote program. Existing procedures can be changed or new ones can be added. More than one version of a remote program can be defined and a version can have more than one procedure defined.

Note: that the program and procedure names are declared with all capital letters. This is not required, but is a good convention to follow. Note also that the argument type is string and not char * as it would be in C. This is because a char * in C is ambiguous. char usually means an array of characters, but it could also represent a pointer to a single character. In RPC language, a null-terminated array of char is called a string.

There are just two more programs to write:

- The remote procedure itself

  Th RPC Version of `printmsg.c`:

```
/*
* msg_proc.c: implementation of the
* remote procedure "printmessage"
*/

#include <stdio.h>
#include "msg.h" /* msg.h generated by rpcgen */

int * printmessage_1(char **msg, struct svc_req *req)

{
   static int result; /* must be static! */
   FILE *f;

   f = fopen("/dev/console", "w");
   if (f == (FILE *)NULL) {
     result = 0;
     return (&result);
    }
   fprintf(f, "%s\n", *msg);
   fclose(f);
   result = 1;
   return (&result);
}
```

  Note that the declaration of the remote procedure `printmessage_1` differs from that of the local procedure printmessage in four ways:

  – It takes a pointer to the character array instead of the pointer itself. This is true of all remote procedures when the `'-' N` option is not used: They always take pointers to their arguments rather than the arguments themselves. Without the '-' N option, remote procedures are always called with a single argument. If more than one argument is required the arguments must be passed in a struct.

- It is called with two arguments. The second argument contains information on the context of an invocation: the program, version, and procedure numbers, raw and canonical credentials, and an SVCXPRT structure pointer (the SVCXPRT structure contains transport information). This information is made available in case the invoked procedure requires it to perform the request.
- It returns a pointer to an integer instead of the integer itself. This is also true of remote procedures when the '-' N option is not used: They return pointers to the result. The result should be declared static unless the '-' M (multithread) or '-' A (Auto mode) options are used. Ordinarily, if the result is declared local to the remote procedure, references to it by the server stub are invalid after the remote procedure returns. In the case of '-' M and '-' A options, a pointer to the result is passed as a third argument to the procedure, so the result is not declared in the procedure.
- An _1 is appended to its name. In general, all remote procedures calls generated by rpcgen are named as follows: the procedure name in the program definition (here PRINTMESSAGE) is converted to all lowercase letters, an underbar (_) is appended to it, and the version number (here 1) is appended. This naming scheme allows multiple versions of the same procedure.

- The main client program that calls it:

```
/*
* rprintmsg.c: remote version
* of "printmsg.c"
*/

#include <stdio.h>
#include "msg.h" /* msg.h generated by rpcgen */
```

```
main(int argc, char **argv)

{
  CLIENT *clnt;
  int *result;
  char *server;
  char *message;

  if (argc != 3) {
     fprintf(stderr, "usage: %s host
     message\n", argv[0]);
     exit(1);
    }

  server = argv[1];
  message = argv[2];

  /*
   * Create client "handle" used for
   * calling MESSAGEPROG on the server
   * designated on the command line.
   */

  clnt = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS, "visible");

  if (clnt == (CLIENT *)NULL) {
   /*
    * Couldn't establish connection
    * with server.
    * Print error message and die.
    */

   clnt_pcreateerror(server);
   exit(1);
   }

  /*
```

```
        * Call the remote procedure
        * "printmessage" on the server
        */

        result = printmessage_1(&message, clnt);
        if (result == (int *)NULL) {
          /*
           * An error occurred while calling
           * the server.
           * Print error message and die.
           */

          clnt_perror(clnt, server);
          exit(1);
         }

        /* Okay, we successfully called
         * the remote procedure.
         */

        if (*result == 0) {

        /*
         * Server was unable to print
         * our message.
         * Print error message and die.
         */

         fprintf(stderr, "%s: could not print your message\n",argv[0]);
         exit(1);
         }

        /* The message got printed on the
         * server's console
         */

         printf("Message delivered to %s\n", server);
         clnt_destroy( clnt );
```

```
    exit(0);
}
```

Note the following about Client Program to Call printmsg.c:

- First, a client handle is created by the RPC library routine `clnt_create()`. This client handle is passed to the stub routine that calls the remote procedure. If no more calls are to be made using the client handle, destroy it with a call to `clnt_destroy()` to conserve system resources.
- The last parameter to `clnt_create()` is visible, which specifies that any transport noted as visible in `/etc/netconfig` can be used.
- The remote `procedure printmessage_1` is called exactly the same way as it is declared in `msg_proc.c`, except for the inserted client handle as the second argument. It also returns a pointer to the result instead of the result.
- The remote procedure call can fail in two ways. The RPC mechanism can fail or there can be an error in the execution of the remote procedure. In the former case, the remote `procedure printmessage_1` returns a `NULL`. In the latter case, the error reporting is application dependent. Here, the error is returned through `*result`.

To compile the remote `rprintmsg` example:

- compile the protocol defined in `msg.x`: `rpcgen msg.x`.

  This generates the header files (`msg.h`), client stub (`msg_clnt.c`), and server stub (`msg_svc.c`).

- compile the client executable:

  ```
  cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
  ```

- compile the server executable:

  ```
  cc msg_proc.c msg_svc.c -o msg_server -lnsl
  ```

The C object files must be linked with the library `libnsl`, which contains all of the networking functions, including those for RPC and XDR.

In this example, no XDR routines were generated because the application uses only the basic types that are included in `libnsl` . Let us consider further what `rpcgen` did with the input file `msg.x`:

- It created a header file called `msg.h` that contained `#define` statements for `MESSAGEPROG, MESSAGEVERS`, and `PRINTMESSAGE` for use in the other modules. This file**must** be included by both the client and server modules.

- It created the client stub routines in the `msg_clnt.c` file. Here there is only one, the `printmessage_1` routine, that was called from the `rprintmsg` client program. If the name of an `rpcgen` input file is `prog.x`, the client stub's output file is called `prog_clnt.c`.

- It created the server program in `msg_svc.c` that calls `printmessage_1` from `msg_proc.c`. The rule for naming the server output file is similar to that of the client: for an input file called `prog.x`, the output server file is named `prog_svc.c`.

Once created, the server program is installed on a remote machine and run. (If the machines are homogeneous, the server binary can just be copied. If they are not, the server source files must be copied to and compiled on the remote machine.)

33.3 Passing Complex Data Structures

`rpcgen` can also be used to generate XDR routines — the routines that convert local data structures into XDR format and vice versa.

let us consider `dir.x` a remote directory listing service, built using `rpcgen` both to generate stub routines and to generate the XDR routines.

The RPC Protocol Description File: `dir.x` is as follows:

```
/*
* dir.x: Remote directory listing protocol
*
* This example demonstrates the functions of rpcgen.
```

```
*/

const MAXNAMELEN = 255; /* max length of directory entry */

typedef string nametype<MAXNAMELEN>; /* director entry */

typedef struct namenode *namelist; /* link in the listing */

/* A node in the directory listing */

struct namenode {
   nametype name; /* name of directory entry */
   namelist next; /* next entry */
  };

/*
* The result of a READDIR operation
*
* a truly portable application would use
* an agreed upon list of error codes
* rather than (as this sample program
* does) rely upon passing UNIX errno's
* back.
*
* In this example: The union is used
* here to discriminate between successful
* and unsuccessful remote calls.
*/

union readdir_res switch (int errno) {
   case 0:
     namelist list; /* no error: return directory listing */
   default:
     void; /* error occurred: nothing else to return */
   };

/* The directory program definition */
```

```
program DIRPROG {
   version DIRVERS {
    readdir_res
    READDIR(nametype) = 1;
   } = 1;
} = 0x20000076;
```

You can redefine types (like `readdir_res` in the example above) using the `struct,` `union`, and `enum` RPC language keywords. These keywords are not used in later declarations of variables of those types. For example, if you define a `union`, `my_un`, you declare using only `my_un`, and not union `my_un`. `rpcgen` compiles RPC unions into C structures. Do not declare C unions using the union keyword.

Running `rpcgen` on `dir.x` generates four output files:

- the header file, `dir.h`,
- the client stub, `dir_clnt.c`,
- the server skeleton, `dir_svc.c` ,and
- the XDR routines in the file `dir_xdr.c`.

This last file contains the XDR routines to convert declared data types from the host platform representation into XDR format, and vice versa. For each RPCL data type used in the `.x` file, `rpcgen` assumes that `libnsl` contains a routine whose name is the name of the data type, prepended by the XDR routine header `xdr_` (for example, `xdr_int`). If a data type is defined in the `.x` file, `rpcgen` generates the required `xdr_` routine. If there is no data type definition in the `.x` source file (for example, `msg.x`, above), then no `_xdr.c` file is generated. You can write a `.x` source file that uses a data type not supported by `libnsl`, and deliberately omit defining the type (in the `.x` file). In doing so, you must provide the `xdr_` routine. This is a way to provide your own customized `xdr_` routines.

The server-side of the `READDIR procedure`, `dir_proc.c` is shown below:

```
/*
```

```
* dir_proc.c: remote readdir
* implementation
*/

#include <dirent.h>
#include "dir.h" /* Created by rpcgen */

extern int errno;

extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(nametype *dirname, struct svc_req *req)

{
  DIR *dirp;
  struct dirent *d;
  namelist nl;
  namelist *nlp;

  static readdir_res res; /* must be static! */

  /* Open directory */
  dirp = opendir(*dirname);

 if (dirp == (DIR *)NULL) {
    res.errno = errno;
   return (&res);
  }

  /* Free previous result */
  xdr_free(xdr_readdir_res, &res);

  /*
   * Collect directory entries.
   * Memory allocated here is free by
   * xdr_free the next time readdir_1
   * is called
   */

   nlp = &res.readdir_res_u.list;
   while (d = readdir(dirp)) {
     nl = *nlp = (namenode *)
     malloc(sizeof(namenode));
     if (nl == (namenode *) NULL) {
       res.errno = EAGAIN;
       closedir(dirp);
       return(&res);
      }
    nl->name = strdup(d->d_name);
    nlp = &nl->next;
  }

  *nlp = (namelist)NULL;

  /* Return the result */
  res.errno = 0;
  closedir(dirp);
  return (&res);
}
```

The Client-side Implementation of implementation of the READDIR procedure, `rls.c` is given below:

```
/*
* rls.c: Remote directory listing client
*/

#include <stdio.h>
#include "dir.h" /* generated by rpcgen */

extern int errno;

main(int argc, char *argv[])
```

```
{
  CLIENT *clnt;
  char *server;
  char *dir;
  readdir_res *result;
  namelist nl;

  if (argc != 3) {
     fprintf(stderr, "usage: %s host
     directory\n",argv[0]);
     exit(1);
   }

  server = argv[1];
  dir = argv[2];

  /*
   * Create client "handle" used for
   * calling MESSAGEPROG on the server
   * designated on the command line.
   */

  cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");

  if (clnt == (CLIENT *)NULL) {
     clnt_pcreateerror(server);
     exit(1);
    }

  result = readdir_1(&dir, clnt);

  if (result == (readdir_res *)NULL) {
     clnt_perror(clnt, server);
     exit(1);
   }
  /* Okay, we successfully called
   * the remote procedure.
   */

  if (result->errno != 0) {
    /* Remote system error. Print
     * error message and die.
     */

    errno = result->errno;
    perror(dir);
    exit(1);
   }

  /* Successfully got a directory listing.
   * Print it.
   */

  for (nl = result->readdir_res_u.list;
       nl != NULL;
       nl = nl->next) {
         printf("%s\n", nl->name);
     }

  xdr_free(xdr_readdir_res, result);
  clnt_destroy(cl);
  exit(0);
}
```

As in other examples, execution is on systems named local and remote. The files are compiled and run as follows:

```
remote$ rpcgen dir.x
remote$ cc -c dir_xdr.c
remote$ cc rls.c dir_clnt.c dir_xdr.o -o rls -lnsl
remote$ cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc -lnsl
remote$ dir_svc
```

When you install `rls` on system local, you can list the contents of `/usr/share/lib` on system remote as follows:

```
local$ rls remote /usr/share/lib
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
local$
```

`rpcgen` generated client code does not release the memory allocated for the results of the RPC call. Call `xdr_free()` to release the memory when you are finished with it. It is similar to calling the `free()` routine, except that you pass the XDR routine for the result. In this example, after printing the list, `xdr_free(xdr_readdir_res, result);` was called.

Note - Use `xdr_free()` to release memory allocated by `malloc()`. Failure to use `xdr_free to()` release memory results in memory leaks.

33.4 Preprocessing Directives

`rpcgen` supports C and other preprocessing features. C preprocessing is performed on `rpcgen` input files before they are compiled. All standard C preprocessing directives are allowed in the `.x` source files. Depending on the type of output file being generated, five symbols are defined by `rpcgen`. `rpcgen` provides an additional preprocessing feature: any line that begins with a percent sign (%) is passed directly to the output file, with no action on the line's content. Caution is required because `rpcgen` does not always place the lines where you intend. Check the output source file and, if needed, edit it.

The following symbols may be used to process file specific output:

`RPC_HDR` — Header file output

`RPC_XDR` — XDR routine output

`RPC_SVC` — Server stub output

`RPC_CLNT` — Client stub output

RPC_TB — Index table output

The following example illustrates tthe use of rpcgens pre-processing features.

```
/*
* time.x: Remote time protocol
*/
program TIMEPROG {
   version TIMEVERS {
     unsigned int TIMEGET() = 1;
    } = 1;
} = 0x20000044;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
% static int thetime;
%
% thetime = time(0);
% return (&thetime);
%}
#endif
```

33.4.1 cpp Directives

rpcgen supports C preprocessing features. rpcgen defaults to use /usr/ccs/lib/cpp as the C preprocessor. If that fails, rpcgen tries to use /lib/cpp. You may specify a library containing a different cpp to rpcgen with the '-' Y flag.

For example, if /usr/local/bin/cpp exists, you can specify it to rpcgen as follows:

```
rpcgen -Y /usr/local/bin test.x
```

33.4.2 Compile-Time Flags

This section describes the `rpcgen` options available at compile time. The following table summarizes the options which are discussed in this section.

Option	Flag	Comments
C-style	`'-' N`	Also called Newstyle mode
ANSI C	`'-' C`	Often used with the -N option
MT-Safe code	`'-' M`	For use in multithreaded environments
MT Auto mode	`'-' A`	-A also turns on -M option
TS-RPC library '	`-' b`	TI-RPC library is default
`xdr_inline` count	`'-' i`	Uses 5 packed elements as default, but other number may be specified

33.4.3 Client and Server Templates

`rpcgen` generates sample code for the client and server sides. Use these options to generate the desired templates.

Flag	Function
`'-' a`	Generate all template files
`'-' Sc`	Generate client-side template
`'-' Ss`	Generate server-side template
`'-' Sm`	Generate makefile template

The files can be used as guides or by filling in the missing parts. These files are in addition to the stubs generated.

33.4.4 Example `rpcgen` compile options/templates

A C-style mode server template is generated from the `add.x` source by the command:

```
rpcgen -N -Ss -o add_server_template.c add.x
```

The result is stored in the file `add_server_template.c`.

A C-style mode, client template for the same `add.x` source is generated with the command line:

```
rpcgen -N -Sc -o add_client_template.c add.x
```

The result is stored in the file `add_client_template.c`.

A make file template for the same `add.x` source is generated with the command line:

```
rpcgen -N -Sm -o mkfile_template add.x
```

The result is stored in the file `mkfile_template`. It can be used to compile the client and the server. If the `'-' a` flag is used as follows:

```
rpcgen -N -a add.x
```

`rpcgen` generates all three template files. The client template goes into `add_client.c`, the server template to `add_server.c`, and the makefile template to m̂akefile.a. If any of these files already exists, `rpcgen` displays an error message and exits.

Note - When you generate template files, give them new names to avoid the files being overwritten the next time rpcgen is executed.

33.5 Recommended Reading

The book *Power Programming with RPC* by John Bloomer, O'Reilly and Associates, 1992, is the most comprehensive on the topic and is essential reading for further RPC programming.

33.6 Exercises

Exercise 33.1 *Use* `rpcgen` *the generate and compile the* `rprintmsg` *listing example given in this chapter.*

Exercise 33.2 *Use* `rpcgen` *the generate and compile the* `dir` *listing example given in this chapter.*

Exercise 33.3 *Develop a Remote Procedure Call suite of programs that enables a user to search for specific files or filtererd files in a remote directory. That is to say you can search for a named file* e.g. file.c *or all files named* `*.c` *or even* `*.x`*.*

Exercise 33.4 *Develop a Remote Procedure Call suite of programs that enables a user to* `grep` *files remotely. You may use code developed previously or unix system calls to implement* `grep`*.*

Exercise 33.5 *Develop a Remote Procedure Call suite of programs that enables a user to* list *the contents of a named remote files.*

Chapter 34

Writing Larger Programs

This Chapter deals with theoretical and practical aspects that need to be considered when writing larger programs.

When writing large programs we should divide programs up into modules. These would be separate source files. `main()` would be in one file, `main.c` say, the others will contain functions.

We can create our own library of functions by writing a *suite* of subroutines in one (or more) modules. In fact modules can be shared amongst many programs by simply including the modules at compilation as we will see shortly..

There are many advantages to this approach:

- the modules will naturally divide into common groups of functions.
- we can compile each module separately and link in compiled modules (more on this later).
- UNIX utilities such as **make** help us maintain large systems (see later).

34.1 Header files

If we adopt a modular approach then we will naturally want to keep variable definitions, function prototypes *etc.* with each module. However what if several modules need to share such definitions?

It is best to centralise the definitions in one file and share this file amongst the modules. Such a file is usually called a **header file**.

Convention states that these files have a `.h` suffix.

We have met standard library header files already *e.g*:

```
#include <stdio.h>
```

We can define our own header files and include then our programs via:

```
#include ``my_head.h''
```

NOTE: Header files usually ONLY contain definitions of data types, function prototypes and C preprocessor commands.

Consider the following simple example of a large program (Fig. 34.1) .

The full listings `main.c, WriteMyString.c` and `header.h` as as follows:

main.c:

```
/*
 * main.c
 */
#include "header.h"
#include <stdio.h>

char *AnotherString = "Hello Everyone";

main()
{
printf("Running...\n");

/*
 * Call WriteMyString() - defined in another file
 */
WriteMyString(MY_STRING);

printf("Finished.\n");
}
```

WriteMyString.c:

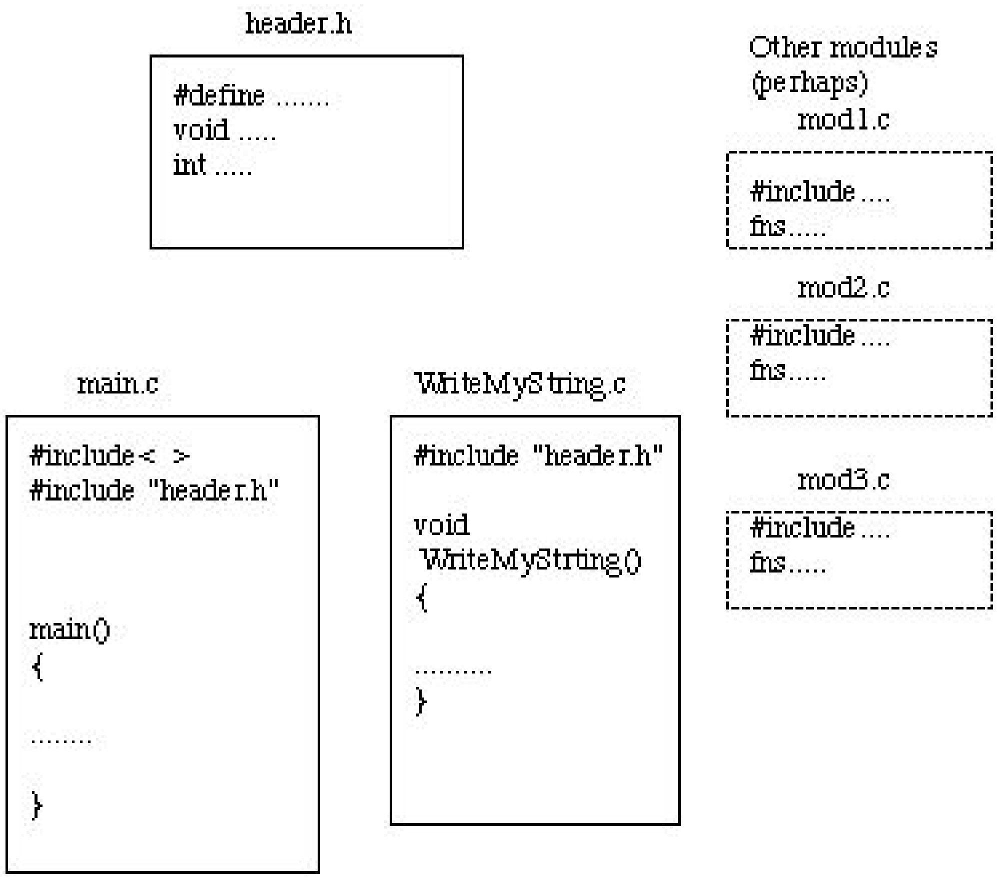


Figure 34.1: Modular structure of a C program

```
/*
 * WriteMyString.c
 */
extern char *AnotherString;

void WriteMyString(ThisString)
char *ThisString;
{
printf("%s\n", ThisString);
printf("Global Variable = %s\n", AnotherString);
}
```

header.h:

```
/*
 * header.h
 */
#define MY_STRING "Hello World"

void WriteMyString();
```

We would usually compile each module separately (more later).

Some modules have a `#include ‘‘header.h’’` that share common definitions.

Some, like *main.c*, also include standard header files also.

`main` calls the function `WriteMyString()` which is in *WriteMyString.c* module.

The function prototype `void` for `WriteMyString` is defined in *Header.h*

NOTE that in general we must resolve a tradeoff between having a desire for each `.c` module to have access to the information it needs solely for its job and the practical reality of maintaining lots of header files.

Up to some moderate program size it is probably best to one or two header files that share more than one modules definitions.

For larger programs get UNIX to help you (see later).

One problem left with module approach:

SHARING VARIABLES

If we have global variables declared and instantiated in one module how can pass knowledge of this to other modules.

We could pass values as parameters to functions, BUT:

- this can be laborious if we pass the same parameters to many functions and / or if there are long argument lists involved.
- very large arrays and structures are difficult to store locally — memory problems with stack.

34.2 External variables and functions

"Internal" implies arguments and functions are defined inside functions — **Local**

"External" variables are defined outside of functions — they are potentially available to the whole program (Global) but **NOT necessarily**.

External variables are always permanent.

NOTE: That in C, all function definitions are external. We CANNOT have embedded function declarations like in PASCAL.

34.2.1 Scope of externals

An external variable (or function) is not always totally global.

C applies the following rule:

The scope of an external variable (or function) begins at its point of declaration and lasts to the end of the file (module) it is declared in.

Consider the following:

```
main()
  { ....  }
```

```
int what_scope;
float end_of_scope[10]

void what_global()
   { ....  }

char alone;

float fn()
   { ....  }
```

main cannot see what_scope or end_of_scope but the functions what_global and fn can. ONLY fn can see alone.

This is also the one of the reasons why we should *prototype* functions before the body of code *etc.* is given.

So here main will not know anything about the functions what_global and fn. what_global does not know about fn but fn knows about what_global since it is declared above.

NOTE: The other reason we *prototype* functions is that some checking can be done the parameters passed to functions.

If we need to refer to an external variable before it is declared or if it is defined in another module we must declare it as an extern variable. *e.g.*

```
extern int what_global
```

So returning to the modular example. We have a global string AnotherString declared in main.c and shared with WriteMyString.c where it is declared extern.

BEWARE the extern prefix is a *declaration* NOT a *definition. i.e* **NO STORAGE** is set aside in memory for an extern variable — it is just an announcement of the property of a variable.

The actual variable must only be defined once in the whole program — you can have as many extern declarations as needed.

Array sizes must obviously be given with
declarations but are not needed with extern declarations. *e.g.*:

```
main.c:    int arr[100]:

file.c:    extern int arr[];
```

34.3 Advantages of Using Several Files

The main advantages of spreading a program across several files are:

- Teams of programmers can work on programs. Each programmer works on a different file.
- An object oriented style can be used. Each file defines a particular type of object as a datatype and operations on that object as functions. The implementation of the object can be kept private from the rest of the program. This makes for well structured programs which are easy to maintain.
- Files can contain all functions from a related group. For Example all matrix operations. These can then be accessed like a function library.
- Well implemented objects or function definitions can be re-used in other programs, reducing development time.
- In very large programs each major function can occupy a file to itself. Any lower level functions used to implement them can be kept in the same file. Then programmers who call the major function need not be distracted by all the lower level work.
- When changes are made to a file, only that file need be re-compiled to rebuild the program. The UNIX make facility is very useful for rebuilding multi-file programs in this way.

34.4 How to Divide a Program between Several Files

Where a function is spread over several files, each file will contain one or more functions. One file will include main while the others will contain functions which are called by others. These other files can be treated as a library of functions.

Programmers usually start designing a program by dividing the problem into easily managed sections. Each of these sections might be implemented as one or more functions. All functions from each section will usually live in a single file.

Where objects are implemented as data structures, it is usual to to keep all functions which access that object in the same file. The advantages of this are:

- The object can easily be re-used in other programs.
- All related functions are stored together.
- Later changes to the object require only one file to be modified.

Where the file contains the definition of an object, or functions which return values, there is a further restriction on calling these functions from another file. Unless functions in another file are told about the object or function definitions, they will be unable to compile them correctly.

The best solution to this problem is to write a header file for each of the C files. This will have the same name as the C file, but ending in .h. The header file contains definitions of all the functions used in the C file.

Whenever a function in another file calls a function from our C file, it can define the function by making a `#include` of the appropriate `.h` file.

34.5 Organisation of Data in each File

Any file must have its data organised in a certain order. This will typically be:

- A preamble consisting of `#define`d constants, `#include`d header files and `typedef`s of important datatypes.
- Declaration of global and external variables. Global variables may also be initialised here.
- One or more functions.

The order of items is important, since every object must be defined before it can be used. Functions which return values must be defined before they are called. This definition might be one of the following:

- Where the function is defined and called in the same file, a full declaration of the function can be placed ahead of any call to the function.
- If the function is called from a file where it is not defined, a prototype should appear before the call to the function.

A function defined as

```
float find_max(float a, float b, float c)
{  /* etc ... ... */
```

would have a prototype of

```
float find_max(float a, float b, float c);
```

The prototype may occur among the global variables at the start of the source file. Alternatively it may be declared in a header file which is read in using a `#include`.

It is important to remember that all C objects should be declared before use.

34.6 The Make Utility

The *make* utility is an intelligent program manager that maintains integrity of a collection of program modules, a collection of programs or a complete system — does not have be programs in practice can be any system of files (*e.g.* chapters of text in book being typeset).

Its main use has been in assisting the development of software systems.

Make was originally developed on UNIX but it is now available on most systems.

NOTE: Make is a programmers utility not part of C language or any language for that matter.

Consider the problem of maintaining a large collection of source files:

```
main.c f1.c .........  fn.c
```

We would normally compile our system via:

```
cc -o main main.c f1.c .........  fn.c
```

However, if we know that some files have been compiled previously and their sources have not changed since then we could try and save overall compilation time by linking in the object code from those files say:

```
cc -o main main.c f1.c ...  fi.o ..  fj.o ...  fn.c
```

We can use the C compiler option (Appendix A) `-c` to create a `.o` for a given module. For example:

```
cc -c main.c
```

will create a `main.o` file. We do not need to supply any library links here as these are resolved at the linking stage of compilation.

We have a problem in compiling the whole program in this *long hand* way however:

• It is time consuming to compile a .c module — if the module has been compiled before and not been altered there is no need to recompiled it. We can just link the object files in. <u>However</u>, it will not be easy to remember which files are in fact up to date. If we link in an old object file our final executable program will be wrong.

• It is error prone and laborious to type a long compile sequence on the command line. There may be many of our own files to link as well as many system library files. It may be very hard to remember the correct sequence. Also if we make a slight change to our system editing command line can be error prone.

If we use the **make** utility all this control is taken care by make. In general only modules that have older object files than source files will be recompiled.

34.7 Make Programming

Make programming is fairly straightforward. Basically, we write a sequence of commands which describes how our program (or system of programs) can be constructed from source files.

The construction sequence is described in
makefiles which contain *dependency rules* and *construction rules.*

A dependency rule has two parts - a left and right side separated by a :

```
left side :  right side
```

The `left side` gives the names of a *target(s)* (the names of the program or system files) to be built, whilst the `right side` gives names of files on which the target depends (eg. source files, header files, data files)

If the *target* is **out of date** with respect to the constituent parts, *construction rules* following the dependency rules are obeyed.

So for a typical C program, when a make file is run the following tasks are performed:

1. The makefile is read. Makefile says which object and library files need to be linked and which header files and sources have to be compiled to create each object file.
2. Time and date of each object file are checked against source and header files it depends on. If any source, header file later than object file then files have been altered since last compilation **THEREFORE** recompile object file(s).
3. Once all object files have been checked the time and date of all object files are checked against executable files. If any later object files will be recompiled.

NOTE: Make files can obey any commands we type from command line. Therefore we can use makefiles to do more than just compile a system source module. For example, we could make backups of files, run programs if data files have been changed or clean up directories.

34.8 Creating a makefile

This is fairly simple: just create a text file using any text editor. The *makefile* just contains a list of file dependencies and commands needed to satisfy them.

Lets look at an example makefile:

```
prog: prog.o f1.o f2.o
  c89 prog.o f1.o f2.o -lm etc.

prog.o: header.h prog.c
  c89 -c prog.c
```

```
f1.o: header.h f1.c
  c89 -c f1.c


f2.o: ---
  ----
```

Make would interpret the file as follows:

1. prog depends on 3 files: `prog.o, f1.o` and `f2.o`. If any of the object files have been changed since last compilation the files must be relinked.
2. prog.o depends on 2 files. If these have been changed prog.o must be recompiled. Similarly for `f1.o` and `f2.o`.

The last 3 commands in the makefile are called *explicit rules* — since the files in commands are listed by name.

We can use *implicit rules* in our makefile which let us generalise our rules and save typing.

We can take

```
f1.o:  f1.c
  cc -c f1.c

f2.o:  f2.c
  cc -c f2.c
```

and generalise to this:

`.c.o:    cc -c $<`

We read this as .source_extension.target_extension: `command`

$< is shorthand for file name with .c extension.

We can put comments in a makefile by using the # symbol. All characters following # on line are ignored.

Make has many built in commands similar to or actual UNIX commands. Here are a few:

`break`	`date`	`mkdir`
`type`	`chdir`	`mv` (move or rename)
`cd`	`rm` (remove)	`ls`
`cp` (copy)	`path`	

There are many more see manual pages for make (online and printed reference)

34.9 Make macros

We can define *macros* in make — they are typically used to store source file names, object file names, compiler options and library links.

They are simple to define, *e.g.*:

```
SOURCES = main.c f1.c f2.c
CFLAGS  = -g -C
LIBS    = -lm
PROGRAM = main
OBJECTS = (SOURCES: .c = .o)
```

where `(SOURCES: .c = .o)` makes .c extensions of SOURCES .o extensions.

To reference or invoke a macro in make do \$(macro_name).*e.g.:*

```
$(PROGRAM) : $(OBJECTS)
$(LINK.C) -o $@ $(OBJECTS) $(LIBS)
```

NOTE:

- `$(PROGRAM) : $(OBJECTS)` – makes a list of dependencies and targets.
- The use of an internal macros *i.e.* `$@`.

There are many internal macros (see manual pages) here a few common ones:

$* — file name part of current dependent (minus .suffix).

$@ — full target name of current target.

$< — .c file of target.

An example makefile for the `WriteMyString` modular program discussed in the above is as follows:

```
#
# Makefile
#
SOURCES.c= main.c WriteMyString.c
INCLUDES=
CFLAGS=
SLIBS=
PROGRAM= main

OBJECTS= $(SOURCES.c:.c=.o)

.KEEP_STATE:

debug := CFLAGS= -g

all debug: $(PROGRAM)

$(PROGRAM): $(INCLUDES) $(OBJECTS)
$(LINK.c) -o $@ $(OBJECTS) $(SLIBS)

clean:
rm -f $(PROGRAM) $(OBJECTS)
```

34.10 Running Make

Simply type `make` from command line.

UNIX automatically looks for a file called `Makefile` (note: capital M rest lower case letters).

So if we have a file called `Makefile` and we type make from command line. The `Makefile` in our current directory will get executed.

We can override this search for a file by typing `make -f make_filename`

e.g. `make -f my_make`

There are a few more `-options` for makefiles — see manual pages.

Chapter 35

Introduction to X/Motif Programming

This book introduces the fundamentals of Motif programming and addreses wider issues concerning the X Window system. The aim of this book is to provide a practical introduction to writing Motif programs. The key principles of Motif programming are always supported by example programs.

The X Window system is very large and this book does not attempt to detail every aspect of either X or Motif. This book is not intended to be a complete reference on the subject.

The book is organised into logical parts, it begins by introducing the X Window system and Motif and goes on to study individual components in detail in specific Chapters. In the remainder of this Chapter we concentrate on why Motif and related areas are important and give a brief history of the development of Motif.

35.1 Why Learn X Window and Motif?

There are many reasons and advantages to programming in X/Motif. A few of these are listed below:

- Motif provides an introduction to graphic user interface (GUI) programming — all computers now employ some form of a GUI to their operating systems and other key applications. Most GUIs adhere to similar design principles. Motif can be as regarded a high level GUI toolkit that adopts and enforces common GUI design principles.

- X Window provides a consistent means of graphical user interaction for UNIX workstations.
- Motif provides a high level toolkit, that already has many fully featured GUI objects. For example cut and paste, multi-line *text editors*, *file browsers*, *drag and drop* mechanisms. Simple yet usable Motif applications can be assembled by *bolting* such objects together. Motif *speeds* up GUI program development.
- The X Window system is device independent — it can run on most common computer platforms. If there is a need for different platforms to interact together over a network, X Window might be a good way to achieve this.
- You may have been using the X Window system and want to understand how the system works.
- Professional X Window programmers are still not that numerous even though they are in great demand. You may be reading this book because you need to learn Motif for this reason. A quick scan through any Computer Vacancies Column in major Computer magazines, journals or employment agencies should highlight this need.

35.2 How to use this book

35.2.1 About this book

The X Window system, or simply X, is very large. It has been through many different versions since its conception although things are now becoming fairly standardised and established.

This book does not attempt to cover all of the X system. We will only look at important parts of the system. Indeed we only study important parts of Motif which is itself a component (albeit a large and significant one) of X. We will where necessary introduce components of X that are not part of Motif. These other components will always be treated as if they are to be used with Motif.

Other important concepts relating to more general Graphical User Interface (GUI) design and programming will be studied. In most cases this is directly in relation to the Motif programming model. However, Motif was designed to adhere to standard GUI design approaches and has guidelines defined in the *Motif Style Guide* (Chapter 52).

35.2.2 Conventions used

X provides functionality via a vast set of subroutine libraries. These may be called from a variety of high level languages.

They are most readily called from C programs as this is the language in which most of X is actually implemented in. We will only give C program examples in the main body of text.

Readers should not be too worried about programming in C. We will not need to get heavily involved in C. Basically we will just be calling X/Motif functions and setting variables and data structures from our C programs. The ANSI C programming conventions are assumed in all examples.

In order to distinguish between program code and other text in this book the following fonts are used:

- All program code, fragments of code are highlighted in a `typewriter` type style.
- Motif and other X data types, structures, variables are also highlighted in a `typewriter` type style.
- Important definitions or concepts are highlighted in *italic* type.

We have already used the notation of X/Motif. For convenience throughout this book, a reference to X is taken to mean the X Window System. We refer to Motif whilst strictly speaking the full title is OSF/Motif where OSF stands for the Open Software Foundation the original developers of Motif.

35.3 Graphical User Interfaces (GUIs)

Before we study Motif in detail it is worth considering why we need GUIs and how they can be effectively designed and used.

35.3.1 Why Use GUIs?

GUIs provide an easy means of data entry and modification. They should provide an attractive and easy to use interface between human and machine. So easy in fact that a non-computer literate person could use the system.

GUI's provide a better means of communication than cumbersome text-based interfaces. Typically, GUIs provide such facilities by means of:

- Extensive use of visual control items — Buttons, menus, icons, scroll bars *etc.*
- Intuitive on screen manipulation of data.
- If a standard GUI is adopted then consistency of use across platforms and applications is afforded. Nearly all MS Windows (or Apple Macintosh) have the same *look and feel* so the learning time for a new application is reduced.
- Multiple applications can be run simultaneously on most machines these days. GUIs provide better screen management of such processes — we can assign one window (or more) to each application.

Although GUIs provide some very powerful advantages, there are a couple drawbacks to the GUI approach:

Efficiency — As we will shortly see even the most basic windowing program can be quite large. This is because we will have to write or call upon many functions to control the windowing system — *e.g.* create, move, resize *etc.* windows; handle input via mouse and keyboard actions; control graphics. Motif was designed as an attempt to reduce such problems by packaging common GUI entities together as *widgets.*

Programming — A different approach to programming is needed from the traditional *command-line approach.* You have probably been used to the top down structured programming approach adhered to by languages such as Pascal and C. We will have to adopt a different strategy known as **event-driven** programming – where the actions in our program will be triggered by mouse and keyboard actions.

35.3.2 Designing GUIs

The subject of Graphical User Interface design is large. Indeed it is a major topic in Computer Science in its own right. Consequently, we could devote the rest of the book to this topic. However many *standard* GUI design rules are prescribed by Motif. Many of these rules are prescribed automatically, whilst others are strongly suggested in the *Motif Style Guide* (Chapter 52). In general it is the low level appearance and operation of specific objects (Buttons, Menus *etc.*) that are automatically facilitated by Motif. The higher level organisation of these objects is left to the control of the application developer. Some perhaps are fairly obvious, though not always strictly adhered to, rules of thumb for GUI design include:

- *Keep the interface as simple as possible* — Do not over clutter a single interface.
- *Keep interfaces as consistent as possible* — Adhere to standard GUI principles.
- *Keep in mind the application user* — Provide easy access to common application interactions and do not over complicate common means of interaction.
- *Allow the user some control of the interface* — This allows some customisation for user preferences.
- *Communicate the application actions to the user* — Maintain a dialogue with the user. For example, do not allow the user believe that the system or application has "hung up" whilst performing intensive computations — display a message or flash an icon to indicate some progress.

The *Motif Style Guide* (Chapter 52) is a valuable source of information in relation to GUI design.

35.4 History of X/Motif

This Section briefy surveys the important stages in the development of GUIs leading the X/Motif approach to GUI design and programming. Once X began to establish itself as one of the prime systems for window programming some issues still remained unresolved until recently, these are also briefly addressed.

35.4.1 Communication before X

Computers first became commercially available in the 1950s. However, they were very large and expensive. They were also hard to program with very little thought was given to human computer interaction. By the 1960s very basic inroads into ease of computer use were being made with the development of more reliable operating systems. This was made possible as computers became smaller in size and more powerful in terms of processing ability. The first seeds of user interaction appeared in the form of early text editors during this period.

The start of the development of GUIs can be traced back to the early 1970's to Alan Kay's research group at Xerox's Palo Alto Research Centre. Two important projects were undertaken there:

- *Dynabook* (Early 1970s) — where the goal was to produce a book sized personal computer with high resolution colour display and a radio link to a worldwide computer network. Mailbox, library, telephone and secretarial functions were also to be incorporated.
- *Star* (Late 1970s) where the goal was to produce a desk-sized personal workstation used by a single person. A high resolution display capable of fast high quality graphics was included. Graphical user interaction was provided by means of a mouse allowing options to be selected from a displayed menu . Later versions of Star introduced *icons* on the screen to represent objects and functions. The idea of *traits* — a characteristic of an object that can be expressed by a set of methods or data and can be applied to, or carried by, the object holding that trait — was also introduced. Traits were later to resurface in last release of Motif (2.0).

The first commercial exploitation of the Xerox work was realised in the early 1980s by Apple, firstly with *Lisa* and then the *Macintosh* series of computers. The Apple GUI proved very successful and popular and by the late 1980s many operating systems had adopted the GUI approach. UNIX vendors such as Sun (with SunView) and Dec (with DEC Windows) and Microsoft with Windows for the PC are examples.

There was one problem with the above developments. Every manufacturer had its own proprietary windowing system. These were all *entirely different* and different systems could not easily communicate with each other. It had been common to have networks of the same computers for some time and it was relatively easy to get machines of the same type in a network configuration to talk to each other.

The X Window system arose out of this very real need. The X system was designed to be platform independent and network-based. With X the programmer can write a single application in a single language and run this program on different machines with little or no modification. Moreover, applications can actually run programs on one computer and have the results displayed on another (or several) computer's terminal. The computer can be a similar model or an entirely different one altogether. The possibilities are endless.

35.4.2 The Motif/Open Look War

Following the creation of the X Window system, two primary high level X interface toolkits came to prominence:

Motif — a product of the Open Software Foundation (OSF), an organization that originally included DEC, IBM and Hewlett-Packard.

Open Look/OpenWindows — a product of Sun and AT&T.

Open Look was designed to support the X Window platform yet still maintain compliance with Sun's older native SunView Window system. Open Look had a slightly different design philosophy to Motif. Indeed for many years Sun claimed that Open Look was superior to Motif. At the time Sun were the substantial market leader vendor of UNIX platforms and therefore had a large influence on such matters. However, recently Sun decided to cease support of Open Look and adopt Motif.

Motif was based on IBM's Common User Access (CUA) guidelines as were both Microsoft Windows and OS/2. Consequently, the visual appearance and mode of operation, the so called *look and feel*, of Motif is similar to that of Microsoft Windows and OS/2. This was a deliberate strategy since there is sound business sense in profiting from an open system. More importantly, however, the predominance of Microsoft Windows in the PC market means that an interface that appears to the user to behave in a similar fashion to Microsoft Windows would be a logical choice in migrating (PC) applications to UNIX. It is probably a mixture of these factors that has led to Sun's decision to stop the development of Open Look and adopt Motif for Sun Workstations.

Following Sun's decision to support Motif, the Common Open Software Environment (COSE) united the major UNIX producers including Sun, DEC, IBM, Hewlett-Packard and UNIX System Laboratories. This has had a significant impact on the endorsement of Motif since it is now the standard choice for UNIX and general cross-platform GUI development. COSE has also prescribed choices of other X libraries concerned with 3D graphics (PEX) and Image extension (XIE) which are closely coupled to Motif.

35.4.3 Versions of Motif

Motif has undergone four major revisions since its conception. Motif 1.0 is now quite old and should probably be avoided as there has been significant upgrades to Motif and the underlying X system in later Motif versions. Motif 1.1 was the next major release but this releases does not support some useful later features, such as drag and drop. However, many applications can still run under Motif 1.1. Motif 1.2 has been available since 1993 and is based on Release 5 of the Xlib and Xt specifications (X11R5). This version of Motif should be in common circulation now.

The latest version of Motif is version 2.0 and it was released in late 1994 as was the latest release of X11 (X11R6). Motif 2.0 provides some significant enhancements and many bug fixes. However, Motif 2.0 is not yet being shipped by the major UNIX vendors. The main reason is that most UNIX vendors made a decision to support a new *Common Desktop Envirnoment* (*see* Section 35.5.2 below) system, CDE 1.0, which uses Motif 1.2 and is not binary compatible with Motif 2.0. It is likely that these companies will not now deliver Motif 2.0 but wait to support the convergence of both products with the newer releases of Motif (2.1) and CDE (1.1). It is expected that both these will become available sometime in 1997.

Even though there have been four major revisions to Motif there have been several minor revisions to Motif. These were mainly fix bugs (sometimes a few hundred at a time !!). However different vendors do not always keep to consistent minor version release numbers.

The subset of Motif addressed in this book has remained more or less untouched by the developments in Motif 2.0. Where there are differences these are highlighted in the text. The major differences that concern us here are the support of additional widgets (Chapter 39 and C++ binding.

The new features of Motif 2.0 are generally concerned with advanced uses of Motif, and are not within the defined scope of this text. All examples in this book have been tested extensively on Motif 1.2 and X11R5. There should be no problem in running these examples under Motif 2.0 or X11R6.

35.5 Culture

In using and writing Motif programs you will inevitably be exposed to key parts of your computer. You will have to write Motif programs in a particlar computer language — usually C or C++. In writing and running Motif applications you will need to interact with the host operating system. You may also need to suitably equip your operating to run or display Motif applications. In conjuction with the operating system, there will be a windowing environment that controls how windows are displayed and managed in general. This section introduces these issues and explains how you configure your system to run Motif applications.

35.5.1 Operating Systems

X Window is designed to be platform independent. Provided that machines which are connected to a network and are suitably equipped with software to run X Window, running applications across any kind of network is possible.

For Unix systems X/Motif is now the only practical window system available. Unix machines will usually be already configured to run some version of X. For users of PCs and Macintosh computers, the standard window environment is Microsoft Windows or the Macintosh Window Interface respectively. For users of these machines there are two options to set up X Window:

X Window/Unix Systems — These basically convert you PC/Macintosh into a X Window/Unix operating system. Many allow the native PC/Mac operating or window system to coexist with the installed Unix system. Many commercial and freeware packages exist for running Unix and the basic X system. Motif libraries have to be purchased to run on top of the basic X system. Probably the most popular system is the freely available Linux system.

Linux is available for both PC and Macintosh (Power PC Macs only). Linux basically converts the host machine into a Unix environment. X/Motif libraries are available for Linux at a small additional cost. For further details on Linux consult the following URLs:

`http://www.linux.org/` — Main Linux site home page.

`http://www.linux.co.uk/` — U.K. Linux Site.

`http://www.rahul.net/kenton/xsites.html` — General X Window and Linux information with many links to related sites

`http://www.mklinux.apple.com/` — Linux on the Macintosh.

There are a few other commercial packages available that run Unix/X Window environments on a PC and Macintosh.

X Window Display/Server Software — Some commercial packages are available that allow the host machine to act as an X server and/or an X display. Some packages act as an X display meaning that X application must be run on a machine that supports an X server but the (X) output of the application can be redirected to X display.

Packages for the PC that allow this include SunSoft Inc.'s SolarNet PC-X 1.1, Hummingbird Communications Ltd. *eXceed 5 for Windows*, Network Computing Devices Inc.'s *PC-Xware 3.0* and Walker Richer and Quinn Inc.'s *Reflection Suite for Windows 5.0.*

Package available to allow a Macintosh to become an X Server include Apple's *MacX*, Intercon's *Planet X*, Netmanage/AGE's *XoftWare*, Tenon *XTen* and White Pine's *eXodus.*

A free X server for PCs and Macintosh exists called *MI/X.*

The WWW site URL:`http://www.rahul.net/kenton/xsites.html` contains links to almost all the above systems.

Note that minimum configurations of many computers in terms of processor speed and/or memory are unlikely to be adequate to support X Window.

35.5.2 The Common Desktop Environment (CDE)

Built on top of the operating system is the windowing environment. This environment controls how windows are displayed and how events invoked by mouse selection, keystrokes *etc.* are processed. The general term for the housekeeping of windows is *window management* and further details on this will be discussed in Section 36.2. X window toolkits, such as Motif and Open Look, generally provide two key components: the window manager and the toolkit libraries.

The window manager basically defines the *look and feel* of a particular toolkit. However, with ever increasing windowing needs, the basic window manager and related system and common application needs have grown into a *desktop environment* providing a whole suite of tools including a window manager. The unification of the Unix and X Window providers with COSE has led to the vendors developing a unified desktop for Unix systems — the *Common Desktop Environment (CDE).* The CDE is built on top of Motif.

The CDE was designed to provide end users with a consistent graphical user interface across workstations and PCs, and software developers with a single set of programming interfaces for platforms that support the X Window System and Motif.

The CDE is intended to:

- Reduce learning time by providing the same appearance and behaviour across multiple operating systems.
- Increase productivity by helping system administrators and end users customize the desktop environment to fit individual work styles and preferences.
- Make learning easier by providing a consistent, rich, and easily accessible context-sensitive on-line help system for help whenever and wherever the user needs it.
- Provide a common set of desktop and application development tools.
- Ease the porting of many existing X Window applications to a new envirronment. Applications should be easy to run accross many different platforms thus reducing the costs of moving to a new environment and helping to protect investments in software.

The CDE core components include:

A login manager — A graphical login screen and manages user access to the system.

A file manager — An on screen graphical file representation where users can directly manipulate icons associated with files to organise the file system and launch applications.

An application manager — The application manager is similar to the file manager except that it is intended to be a user specific list of files.

A session manager — Users can easily customize their environment.

The CDE window manager — The control mechanism for the visual user interface, or desktop, of a session. The CDE window manager includes a FrontPanel and a workspace manager. A user can manage all aspects of a session (except the initial login) through objects on the FrontPanel.

An inter-application messaging system — This aims to provide facilitates for the seamless interaction between applications.

A desktop tool set — A comprehensive set of productivity tools including multimedia-enabled mail, text editor, calendar, clock and icon editor, are provided with the CDE.

Application development tools — A comprehensive set of development tools including debuggers, application manager, application (Motif GUI) builder are provided with the CDE.

Application integration components — Aplications written on any X Window system or toolkit should be easy to integrate with the CDE tools provided.

Section 1 addresses the CDE from a user's perspective.

35.5.3 C/C++ programming

This books assumes a knowledge of ANSI/ISO C. All program examples are written in C. C++ is also a popular language for writing X Window programs. C++ actually supports ANSI/ISO C, so some minor modifications are all that is required to convert the C examples provided in this text to C++ (see Chapter 59). Motif 2.0 actually provides C++ support built in.

In order to be most productive in writing Motif it is advisable that you should have at your disposal standard C program developments tools (such as good compilers, editors, debugging tools *etc*). Section 38.5 discusses compilation issues further.

35.6 Religion

Throughout the development of X/Motif some key companies and consortia have been repsonsible for key aspects of the system. We briefly summarise these contibutions in this Section. Another key aspect to the development of Motif is the prescribed *uniformity* of Motif applications. The *Motif Style Guide* is the main reference to Motif application development.

35.6.1 OSF, X Consortium, Open Group

The Open Software Foundation consortium provides many services and is not solely concerned with X Window related matters. The OSF licenses Motif, offers training courses, testing and certification of software intended for commercial use.

The X Consortium distributes the X System and manuals at a minimal cost (basically the cost of distribution media and shipping). Motif is built on top of the X System.

The Open Group was formed in February, 1996 by the consolidation of the two leading open systems consortia, X/Open Company Ltd. and the Open Software Foundation (OSF). Under the Open Group umbrella, OSF and X/Open work together to deliver technology innovations and wide-scale adoption of open systems specifications. From the beginning of 1997 the Open Group will have responsibility for the X Window System transferred from the X Consortium.

For further information, the OSF, X Consortium and the Open Group can be contacted at the following addresses:

Open Software Foundation, 11 Cambridge Center Cambridge MA 02142. Tel (USA): 617/621-8700. Email: info@osf.org WWW:
http://www.osf.org/

X Consortium Inc., One Memorial Drive PO BOX 546 Cambridge MA 02142-0004. Tel (USA):617/374-1000. WWW: http://www.x.org/

Open Group, 11 Cambridge Center, Cambridge, MA 02142: Tel (USA): 617/621-7300. Email: info@opengroup.org WWW: http://www.opengroup.org/

35.6.2 Motif and COSE

The Common Open Software Environment (COSE) was formed when the major UNIX producers, including Sun, DEC, IBM, Hewlett-Packard and UNIX system Laboratories, decided to unite and attempt to standardise UNIX implementations worldwide in 1993. The remit of COSE is to define standard cross-platform UNIX systems incorporating application program interfaces, windows interfacing, desktop environments, graphics, multimedia, system management, support for distributed computing and large scale data management. The standardisation of a Common Desktop Environment (CDE) for UNIX resulted in the adoption of Motif for this purpose.

As such, the OSF, Open Group, and COSE can be regraded as the *elders* of Motif and future releases of Motif will be determined by them.

In late 1995, The OSF announced the formal signing of the Joint Development agreement for the further enhancement and evolution of the Common Desktop Environment and OSF/Motif under the OSF Prestructured Technology (PST) development process.

35.6.3 Motif Style Guide

The *Motif Style Guide* can be regarded as the *Bible* for Motif Application developers. Along with a Motif Reference Manual and a programming text book, the *Motif Style Guide* provides an invaluable source of information.

The *Motif Style Guide* provides a set of guidelines that provides a framework for the behaviour of Motif application developers, GUI developers, widget developers and window managers. The basic idea is that all Motif applications that adhere to the prescribed style will maintain a high level of consistency. Also, since Motif follows CUA guidelines, Motif applications will be similar to Microsoft Windows applications in terms of appearance and user operation. For developers of commercial applications, the adoption of Motif style is critical.

Many standard GUI issues are integrated into a Motif Widget default behaviour. Therefore, these defaults should only be modified with great care and consideration for such implications. Other aspects of style are left to the developer. The *Motif Style Guide* only *suggests* certain standard operations.

As has been mentioned, the *Motif Style Guide* has a greater scope than just the Motif application developer (the intended audience of this text). Chapter 52 summarises many important issues relating to the application developer. Where appropriate, specific style considerations are mentioned elsewhere in this text.

Chapter 36

The X Window System Environment

36.1 What is the X Window system?

The *X Window* system provides a way of writing device independent graphical and windowing software that can be easily ported from machine to machine. X is a network-based system and supports cross-platform communication — we can get different machines to talk to each other.

At the highest level of X there are two basic features: The *window manager* and the *toolkit*. The window manager (wm) controls GUI aspects such as appearance of windows and interaction with the user. The toolkits are C subroutine libraries where we describe how to contsruct the GUI and how to attach the GUI to the remainder of the application. Motif is one such toolkit. Motif is built on other toolkits in the X System: Xt Intrinsics, an intermediate level toolkit, and the low level X Library (Xlib). The relevance of these will be made apparent in due course.

36.1.1 X Window Principles

All forms of displaying of information in X are *bit-mapped* which means that every pixel on the screen is individually controllable. Therefore we can draw pictures and use text. The requirement for bit-mapped graphics means that high quality monitors are necessary. However, most computer systems provide monitors of this type.

X, like most other windowing systems, divides the screen into various parts that control input and output. Each part is called a *window*. A window can have many uses. A window can display graphics, receive input from a mouse, act as a standard terminal (*e.g.* an Xterm — a standard text based terminal emulation window) *etc.*.

Not all applications need to consist of a single window. We can have many windows associated with different parts of one application. Each subwindow is called a *child* and it usually remains under the control of it's *parent* window.

There is one special window, the background or *root* window. All other windows are children of the root.

36.2 The Window Manager

The *window manager* is responsible for manipulating windows on the screen. The window manager performs the following operations:

- Placement and movement of windows.
- Resizing of windows.
- Iconification of windows — how the window appears when the window is *minimised*.
- Starting and manipulation of windows.
- Control of input to windows.

Controlling the window environment is not easy and has many facets. For instance, there may be multiple applications running simultaneously and a conflict may arise for input:

Does a keyboard input go in a window where the mouse currently points or must a window be explicitly chosen?

The window manager is also predominantly responsible for the appearance and user interaction (the *look and feel*) of the interface. Since the development of X, there have been a few different window managers. The look and feel varies a lot between each window manager. The Motif window manager (*mwm*) is probably the window manager you are most familiar with as this comes with the Motif system. Other window managers include Sun's

Open Look window manager (*olwm*) and the Tab or Tom's window manager (*twm*).

Most major Unix vendors now supply the CDE which by default runs the desktop window manager, *dtwm*. The development of the common desktop will probably result in *dtwm* superseding *mwm*. The CDE default window manager can easily be altered to run the above altenative window managers if desired. Since the CDE is built on top of motif there are many similarities between *dtwm* and *mwm* (*see* Section 1.8 below).

The Motif *look and feel*, as defined by the *Motif Style Guide*, is basically enforced by *mwm* or *dtwm*. Consequently, interaction between applications and these window managers are eased if the applications obey standard Motif/CDE guidelines.

Chapter 37

The X Window Programming Model

This Chapter introduces the basic concepts and principles that are of concern to the Motif programmer. We define basic system concepts, describe the three basic levels of the X programming model and describe basic Motif components.

37.1 X System Concepts and Definitions

X requires a system that consists of workstations capable of bit-mapped graphics. These can be colour or monochrome.

A *display* is defined as a workstation consisting of a keyboard, a pointing device (usually a mouse although it could be a track ball or graphics tablet, for instance) and one or more screens.

37.1.1 Clients and Servers

X is network oriented and applications need not be running on the same system as the one supporting the display. This can sometimes be quite complicated for a system such as X to manage and so the concept of *clients* and *servers* was introduced.

You need not worry too much about the practicality of this, as normally X makes this transparent to the user — especially if we run programs on a single workstation. However, in order to fully understand the workings of X, some notion of these concepts is required.

The program that controls each display is known as the *server*. This terminology may seem a little odd as we may be used to the server as something across the network such as a file server. *Here*, the server is a local program that controls our *display*. Also our display may be available to other systems across the network. In this case our system does act as a true display server.

The server acts as a go-between between user programs, called *clients* or applications and the resources of the local system. These run on either local or remote systems.

Tasks the server performs include:

- allowing access by multiple clients,
- interpreting network messages from clients,
- two-dimensional graphics display,
- maintain local resources such as windows, cursors, fonts and graphics.

37.2 The X Programming Model

The client and server are connected by a communication path called (*surprise, surprise*) the *connector*. This is performed by a low-level C language interface known as *Xlib*. Xlib is the lowest level of the X system software hierarchy or architecture (Fig 37.1). Many applications can be written using Xlib alone. However, in general, it will be difficult and time consuming to write complex GUI programs only in Xlib. Many *higher level* subroutine libraries, called **toolkits**, have been developed to remedy this problem.

Note: X is not restricted to a single language, operating system or user interface. It is relatively straightforward to link calls to X from most programming languages. An X application must only be able to generate and receive messages in a special form, called *X protocol messages*. However, the protocol messages are easily accessible as C libraries in Xlib (and others).

There are usually two levels of toolkits above Xlib

- **X Toolkit (Xt) Intrinsics** are parts of the toolkit that allow programmers to build new widgets.

- **Third Party Toolkits** — such a Motif.

An application program in X will usually consist of two parts. The graphical user interface written in one or more of Xlib, Xt or Motif and the algorithmic or functional part of the application where the input from the interface and other processing tasks are defined. Fig. 37.1 illustrates the relationships between the application program and the various parts of the X System.

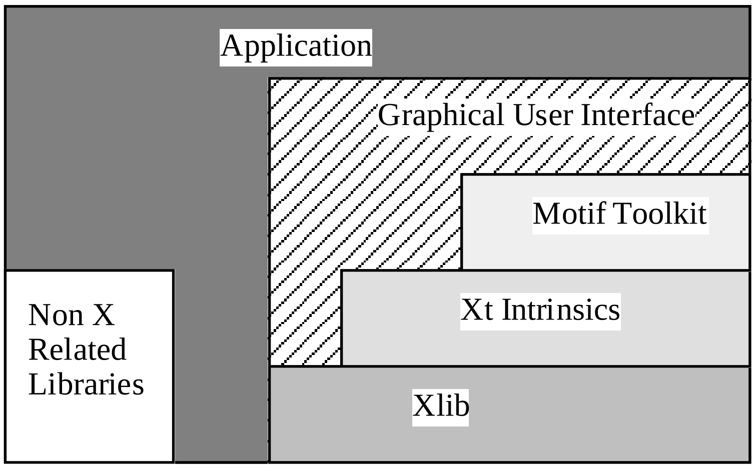


Figure 37.1: The X Programming Model

The main concern of this text is to introduce concepts in building the graphical user interface in X and Motif in particular. We now briefly describe the main tasks of the three levels of the X programming model before embarking on writing Motif programs.

37.2.1 Xlib

The main task of Xlib is to translate C data structures and procedures into the special form of X protocol messages which are then sent off. Obviously the converse of receiving messages and converting them to C structures is performed as well. Xlib handles the interface between client (application) and the network.

37.2.2 Xt Intrinsics

Toolkits implement a set of user interface features or application environments such as menus, buttons or scroll bars (referred to as **widgets**).

They allow applications to manipulate these features using object-oriented techniques.

X Toolkit Intrinsics or *Xt Intrinsics* are a toolkit that allow programmers to create and use new widgets.

If we use widgets properly, it will simplify the X programming process and also help preserve the *look and feel* of the application which should make it easier to use.

We will have to call some Xt functions when writing Motif programs since Motif is built upon Xt and thus needs to use Xt. However, we do notneed to fully understand the workings of Xt as Motif takes care of most things for us.

37.2.3 The Motif Toolkit

X allows extensions to the *Xt Intrinsics* toolkit. Many software houses have developed custom features that make the GUI's appearance attractive, easy to use and easy to develop. *Motif* is one such toolkit.

The third party toolkits usually supply a special client called the **window manager**

37.3 Currency

The basic unit of currency of Motif is the *widget.* The widget is the basic building block for the GUI. It is common and beneficial for most GUIs assembled in Motif to look and behave in a similar fashion. Motif enforces many of these features by providing default actions for each widget. Motif also prescribes certain other actions that should, whenever possible, be adhered to. Information regarding Motif GUI design is provided in the *Motif Style Guide.* We now briefly address general issues relating to Motif widgets and style.

37.3.1 Widget Classes and Hierarchies

A *widget*, in Motif, may be regarded as a general abstraction for user-interface components. Motif provides widgets for almost every common GUI component, including buttons, menus and scroll bars. Motif also provides widgets whose only function is to control the layout of other widgets — thus enabling fairly advanced GUIs to be easily designed and assembled.

A widget is designed to operate independently of the application except through well defined interactions, called *callback functions*. This takes a lot of mundane GUI control and maintenance away from the application programmer. Widgets know how to redraw and highlight themselves, how to respond to certain events such as a mouse click *etc.* Some widgets go further than this, for example the Text widget is a fully functional text editor that has built in cut and paste as well as other common text editing facilities.

The general behaviour of each widget is defined as part of the Motif (Xm) library. In fact Xt defines certain base classes of widgets which form a common foundation for nearly all Xt based widget sets. Motif provides a *widget set*, the Xm library, which defines a complete set of widget classes for most GUI requirements on top of Xt (Fig 37.1).

The Motif Reference Manual provides definitions on all aspects of widget behaviour and interaction. Basically, each widget is defined as a C data structure whose elements define a widget's data attributes, or *resources* and pointers to functions, such as *callbacks*.

Each widget is defined to be of a certain *class*. All widgets of that class inherit the same set of resources and callback functions. Motif also defines a whole hierarchy of widget classes. There are two broad Motif widget classes that concern us. The *Primitive* widget class contains actual GUI components, such as buttons and text widgets. The *Manager* widget class defines widgets that hold other widgets.

Chapter 38 introduces basic Motif widget programming concepts and introduces how resources and callback functions are set up. Chapter 39 then goes on to fully define each widget class and the Motif widget class hierarchy. Following Chapters then address each widget class in detail.

37.3.2 Motif Style — GUI Design

The *Motif Style Guide* should be read by every Motif Application developer. The Style Guide is not intended to be a Motif programming manual. This book is not intended to be a complete guide to Motif style. The books should be regarded as essential companions along with a good Motif reference source.

The *Motif Style Guide* provides a set of guidelines that specify a framework for the behaviour of Motif application developers, GUI developers, widget developers and window managers. Many standard GUI design and behaviour issues are integrated into a Motif widget's default settings. Therefore these defaults should only be modified with great care and consideration. Other aspects of style are left to the developer. The *Motif Style Guide* only suggests certain standard operations, but where appropriate these should be adopted.

The *Motif Style Guide* prescribes many common forms of interaction and interface design. For example, it defines how menus should be constructed, used and organised. Chapter 52 summarises all the common style concerns for the Motif programmer. Where appropriate specific reference is made in individual Sections to Motif style for a particular widget.

Chapter 38

A First Motif Program

In this Chapter we will develop our first Motif program. The main purpose of the program is to illustrate and explain many of the *fundamental* steps of nearly every Motif program.

38.1 What will our program do?

Our program, `push.c`, will create a window with a single push button in it. The button contains the string, "Push Me". When the button is pressed (with the *left* mouse button) a string is printed to *standard output*. We are not yet in a position to write back to any window that we have created. Later Chapters will explore this possibility. However, this program does illustrate a simple interface between Motif GUI and the application code.

The program also runs forever. This is a key feature of event driven processing. For now we will have to quit our programs by either:

- Using the Operating System to terminate the program (process) — there are many ways in which this could be done. The easiest way is to use `ctrl-c` to quit from the command line.

- Use the *Window Menu* quit option (Section 1.8) — depress *right mouse* down around the top perimeter of the window and choose the *quit* option from menu. The *OSF/Motif Style Guide*(Chapter 52), in common with standard GUI practice, also prescribes that *hot keys*, or *keyboard shortcuts*, or *mnemonics* should be facilitated so that common actions can be performed from the keyboard. It is standard convention that the **Meta-F4** is used to close an application.

In forthcoming Chapters (see also Exercise 38.1) we will see how to quit the program from within our programs.

The display of `push.c` on screen will look like this:

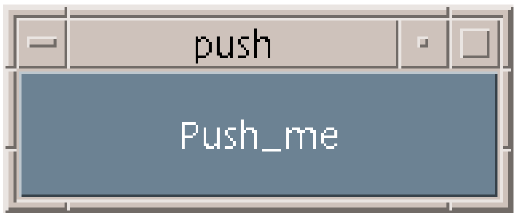


Figure 38.1: Push.c display

38.2 What will we learn from this program?

As was previously stated, the main purpose of studying the program is to gain a fundamental understanding of Motif programming. Specifically the lessons that should be clear before embarking on further Motif programs are:

- How to write and compile a Motif Program.
- The relationship between Xlib, Xt Intrinsics and Motif.
- How to create simple widgets and manage its resources.
- How widgets handle events.
- How to call functions from events — the Interface between the Motif GUI and the application code.

We now list the complete program code and then go on to study the code in detail in the remainder of this Chapter.

38.3 The push.c program

The complete program listing for the push.c program is as follows:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

/* Prototype Callback function */

void pushed_fn(Widget , XtPointer ,
               XmPushButtonCallbackStruct *);


main(int argc, char **argv)

{   Widget top_wid, button;
    XtAppContext  app;

    top_wid = XtVaAppInitialize(&app, "Push", NULL, 0,
        &argc, argv, NULL, NULL);

    button = XmCreatePushButton(top_wid, "Push_me", NULL, 0);

    /* tell Xt to manage button */
XtManageChild(button);

/* attach fn to widget */
    XtAddCallback(button, XmNactivateCallback, pushed_fn, NULL);

    XtRealizeWidget(top_wid); /* display widget hierarchy */
    XtAppMainLoop(app); /* enter processing loop */

}

void pushed_fn(Widget w, XtPointer client_data,
               XmPushButtonCallbackStruct *cbs)
  {
     printf("Don't Push Me!!\n");
  }
```

38.4 Calling Motif, Xt and Xlib functions

When writing a Motif program you will invariably call upon **both** Motif and Xt functions and data structures *explicitly*. You will not always call Xlib functions or structures explicitly (but recall that Motif and Xt are built upon Xlib and they may call Xlib function from their own function calls).

In order to distinguish between the various toolkits, X adopts the following convention:

- Motif function and data structure names begin with **Xm**. So in `push.c`: `XmStringCreateSimple()` and `XmStringFree()` belong to Motif toolkit.
- Xt Intrinsics functions and most data structures begin with **Xt**. *e.g.* `XtVappInitialize()` and `XtVaCreateManagedWidget()`. The `Widget` data structure is an exception to this rule.
- Xlib functions and most data structures begin with **X**. There are no Xlib functions used in `push.c`. An example of an Xlib function call is `XDrawString()`.

38.4.1 Header Files

In order to be able to use various Motif, Xt Intrinsics or Xlib data structures we must include header files that contain their definitions.

The X system is very large and there are many header files. Motif header files are found in `#include<Xm/...>` subdirectories, the Xt and Xlib header files in `#include<X11/...>` subdirectories (*E.g.* the Xt Intrinsic definitions are in `#include<X11/Intrinsics.h>`).

Every Motif widget has its own header file, so we have to include the `<Xm/PushB.h>` file for the push button widget in `push.c`.

We do not have to explicitly include the Xt header file as `<Xm/Xm.h>` does this automatically. Every Motif program will include `<Xm/Xm.h>` — the general header for the motif library.

38.5 Compiling Motif Programs

To compile a Motif program we have to link in the Motif, Xt and Xlib libraries. To do this use: `-lXm -lXt -lX11` from the compiler command line. **NOTE:** The order of these is important.

So to compile our `push.c` program we should do:

```
cc push.c -o push -lXm -lXt -lX11
```

The exact compilation of your Motif programs may require other compiler directives that depend on the operating system and compiler you use. You should *always* check your local system documentation or check with your system manager as to the exact compilation directives. You should also check your C compiler documentation. For example you may need to specify the exact path to a nonstandard location of include (`-I flag`) or library (`-L flag`) files. Also our `push.c` program is written with ANSI style function calls and some compilers may require this knowledge explicitly. Some implementations of X/Motif do not strictly adhere to the ANSI C standard. In this case you may need to turn ANSI C function prototyping *etc.* off.

Having successfully complied you Motif program the command:

```
push
```

should successfully run the program and display the PushButton on the screen.

38.6 Basic Motif Programming Principles

Let us now analyse the `push.c` in detail. There are six basic steps that nearly all Motif programs have to follow. These are:

1. Initializing the toolkit
2. Widget creation
3. Managing widgets
4. Setting up events and callback functions
5. Displaying the widget hierarchy
6. Enter the main event handling loop

We will now look at each of these steps in detail.

38.6.1 Initialising the toolkit

The initialisation of the Xt Intrinsics toolkit must be the first stage of any basic Motif program.

There are several ways to initialise the toolkit. `XtVaAppInitialize()` is one common method. For most of our programs this is the only one that need concern us.

When the `XtVaAppInitialize()` function is called, the following tasks are performed:

- The application is connected to the X display.
- The application is parsed for the standard X command-line arguments.
- Resources are set up.
- A top level window is created — this is returned by the function call to the `Widget` data structure `top_wid` in `push.c`.

`XtVaAppInitialize()` has several arguments:

Application context (address of) — This is a structure that Xt requires for operation. For the Motif programs that we will be considering we do not need to know anything about this, except the need to set it in our program.

Application class name — A string that is used to reference and set resources common to the application of even a collection of applications. Chapter 40 deals with many resource setting mechanisms that uses this class name. In these coming examples note that the class name has been set to the string, "Push".

Command line arguments — The third and fourth arguments specify a list of objects of the special X command line arguments that can be specified to an X program. The third argument is the list, the fourth the number in the list. This is advanced X use and is not considered further in this text. Just set the third argument to `NULL` and the fourth to 0. The fifth and sixth arguments `&argc` and `argv` contain the values of any command line argument given. These arguments may be used to receive command line input of data in standard C fashion (*e.g.* filenames for the program to read). Note that the command line may be used (Section 40.4) to set certain resources in X. However these will have been removed from the `argv` list if they have been correctly parsed and acted upon before being passed on to the remainder of the program.

Fallback Resources — Fallback resources provide security against errors in other setting mechanisms. Fallback resources are ignored, if resources are set by any other means. Chapter 40 deals with many resource setting mechanisms and Section 40.7 gives examples of setting fallback resources. A fallback resource is a `NULL` terminated list of `Strings`. For now we will simply set it to `NULL` as no fallback resources have been specified.

Additional Parameters — a `NULL` terminated list. These are also used only for advanced applications, so we will set them to `NULL`.

38.6.2 Widget Creation

There are several ways to create a widget in Motif:

- There is a specific function for creating each widget.
- There are several convenience functions for *generic* widget creation and even creating and managing widgets with a single function call.

We will introduce the convenience functions shortly but for now we will continue with the simpler first method of widget creation.

In general we create a widget using the function:

```
XmCreate<widget name>().
```

So, to create a push button widget we use `XmCreatePushButton()`
Most `XmCreate<widget name>()` functions take 4 arguments:

- The parent widget — `top_wid` in `push.c`.
- The name of the created widget — a string – `"Push_Me"` – in `push.c`.
- Command line / Resource list — `NULL` in `push.c`.

- The number of arguments in the list.

The argument list can be used to set widget resources (height, width *etc.*) at creation. The name of the widget may also be important when setting a widget's resources. The actual resources set depend on the class of the widget created. The individual Chapters and reference pages on specific widgets list widget resources. Chapter 40 deals with general issues of setting resources and explores the methods described here further.

38.7 Managing Widgets

Once a widget has been created it will usually want to be managed. `XtManageChild()` is a function that performs this task.

When this happens all aspects of the widget are placed under the control of its parent. The most important aspect of this is that if a widget is left unmanaged, then it will remain *invisible* even when the parent is displayed. This provides a mechanism with which we can control the on screen visibility of a widget — we will look at this in more detail in Chapter 43. Note that if a parent widget is not managed, then a child widget will remain invisible *even if* the child is managed.

Let us leave this topic by noting that we can actually create and manage a widget in one function called `XtVaCreateManagedWidget()`. This function can be used to create any widget. We will meet this function later in the Chapter 40.

38.8 Events and Callback Functions

38.8.1 Principles of Event Handling

When a widget is created it will automatically respond to certain internal events such as a window manager request to change size or colour and how to change appearance when pressed. This is because Xt and Motif frees the application program from the burden of having to intercept and process most of these events. However, in order to be useful to the application programmer, a widget must be able to be easily attached to application functions.

Widgets have special *callback functions* to take care of this.

An *event* is defined to be any mouse or keyboard (or any input device) action. The effect of an event is numerous including window resizes, window repositioning and the invoking functions available from the GUI.

X handles events *asynchronously*, that is, events can occur in any order. X basically takes a continuous stream of events and then dispatches them according to the appropriate applications which then take appropriate actions (remember X can run more than one program at a time).

If you write programs in Xlib then there are many *low level* functions for handling events. Xt, however, simplifies the event handling task since widgets are capable of handling many events for us (*e.g.* widgets are automatically redrawn and automatically respond to mouse presses). How widgets respond to certain actions is predefined as part of the widget's resources. Chapter 50 gives a practical example of changing a widget's default response to events.

38.8.2 Translation tables

Every widget has a *translation table* that defines how a widget will respond to particular events. These events can enable one or more actions. **Full** details of each widgets response can be found in the Motif Reference material and manuals.

An example of part of the translation table for the push button is:

```
BSelect Press: Arm()
BSelect Click: Activate(), Disarm()
```

`BSelect Press` corresponds to a left mouse pressed down and the action is the `Arm()` function being called which cause the display of the button to appear as it was depressed. If the mouse is clicked (pressed and released), then the `Activate()` and `Disarm()` functions are called, which will cause the button to be reactivated.

Keyboard events can be listed in the table as well to provide facilities such as *hot keys*, function/numeric select keys and help facilities. These can provide short cuts to point and click selections.

Examples include: `KActivate` — typically the return key, `KHelp` — the HELP or F1 key.

38.8.3 Adding callbacks

The function `Arm()`, `Disarm()` and `Activate()` are examples of predefined *callback functions.*

For any application program, Motif will only provide the GUI. The Main body of the application will be attached to the GUI and functions called from various events within the GUI.

To do this in Motif we have to add our own callback functions.

In `push.c` we have a function `pushed_fn()` which prints to standard output.

The function `XtAddCallback()` is the most commonly used function to attach a function to a widget.

It has four arguments:

- The widget in which the callback is to be installed, `button` in our example.
- The name of the callback resource. In our example we set `XmNactivateCallback`.
- The pointer to the function to be called.
- Client data that may get passed to the callback function. Here we do not pass any data and it is therefore set to `NULL`.

In addition to performing a job like highlighting the widget, each event action can also call a program function. So, we can also hang functions off the `arm`, `disarm` *etc.* actions as well. We use `XmNarmCallback`, `XmNdisarmCallback` names to do this.

So, if we wanted to attach a function `quit()` to a `disarm` for the `button` widget, we would write:

```
    XtAddCallback(button, XmNdisarmCallback, quit, NULL);
```

38.8.4 Declaring callback functions

Let us now look at the declaration of the application defined callback function. All callback functions have this form.

```
void pushed_fn(Widget w,
               XtPointer client_data,
               XmPushButtonCallbackStruct *cbs)
```

The first parameter of the function is the widget associated with the function (`button` in our case).

The second parameter is used to pass client data to the function. We will see how to attach client data to a callback later. We do not use it in this example so just leave it defined as above for now.

The third parameter is a pointer to a structure that contains data specific to the particular widget that called the function and also information about the event that triggered the call.

The structure we have used is a `XmPushButtonCallbackStruct`. A *Callback Structure* has the following general form:

```
typedef struct {
     int reason;
     XEvent *event;
     .... widget specifics ... } Xm<widget>CallbackStruct;
```

The `reason` element contains information about the callback such as whether `arm` or `disarm` invoked the call and the `event` element is a pointer to an (Xlib) XEvent structure that contains information about the event.

38.9 Finishing off — displaying widgets and event loops

We have nearly finished our first program. We have two final stages to perform which every Motif program has to perform. That is to tell X to:

- Display or **realize** the widgets. This is achieved via the `XtRealizeWidget()` function. In `push.c` we pass the top level widget `top_wid` to the function so that all child widgets are displayed.

- Enter the main event handling loop. The function `XtAppMainLoop(app)` does this. After this call, Xt has control over the program and it is this that dispatches events that invoke callbacks *etc.* **Note:** The application code will be idle until a user activates an event.

38.10 Exercises

Exercise 38.1 *Write a Motif program that displays a button labelled "Quit" which terminates the program when the button is depressed with the left mouse button.*

Chapter 39

Widget Basics

In the last Chapter we introduced some basic Motif programming concepts and introduced one specific widget, the PushButton, used in the example program developed. There are many other classes of widgets defined in Motif. Motif widgets are provided to perform a wide variety of common GUI tasks. This Chapter overviews the classes of Motif widgets. Following Chapters go on to study individual widgets in detail.

39.1 Widget Classes

The organisation of a large system of GUI components in Motif can be quite complex. In order to aid the design and understanding of Motif various widget classes have been constructed. Each class can be categorised by a broad functionality, at a variety of levels. Motif defines a hierarchy of widget classes (Fig 39.1) and a widget will *inherit* properties from a higher class in the widget hierarchy (Section 39.5). Some levels of the hierarchy have strong relationships with Xt Intrinsics since widgets are actually created in this toolkit.

The levels of the hierarchy and a broad functionality of each level are:

Core — The top of the hierarchy comes from Xt intrinsics. This *superclass* defines background, size and position properties that are common to all widgets.

XmPrimitive — The superclass of all primitive widgets. Primitive widgets are the basic building blocks of any Motif GUI (See Section 39.1.3

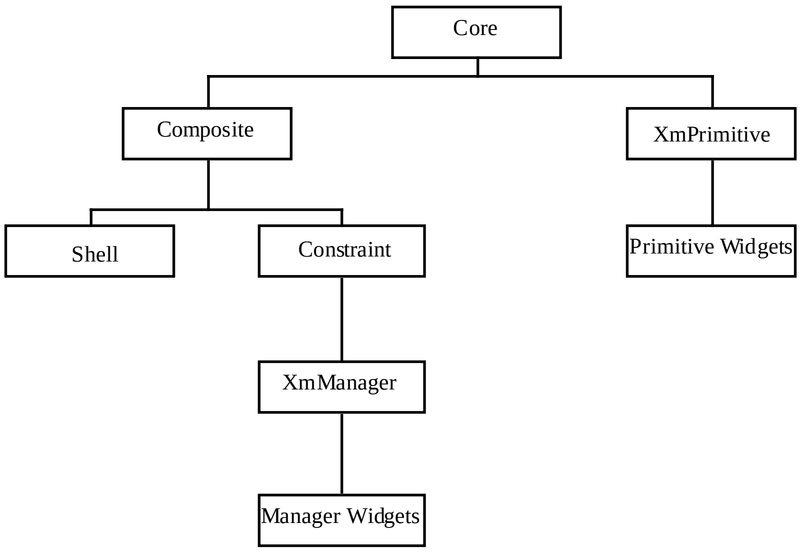


Figure 39.1: Widget Hierarchy

below).

Composite — The superclass of containers for widgets. Two sub-classes of the *Composite* class are defined:

Shell widgets control the interfacing of Motif with the window manager.

Constraint widgets are concerned with the organisation (positioning, alignment, *etc.*) of widgets contained within them. *XmManager* is a subclass of the constraint widget class. Manager widgets are discussed further in Section 39.1.3 below.

39.1.1 Shell Widgets

All widgets are contained in a shell widget. This is usually the top level widget. The primary function of a shell widget is as an interface to the window manager.

The **application** shell is normally the top level for an application and is created by `XtVaAppInitialze()` or related functions.

There are two other shells:

- The **Override shell** is used for pop-up menus that must be at the top-level. To achieve this, the window manager must usually be bypassed. Consequently, the override shell is not often used by Motif programs.

- The **Transient** shell is used for dialogs. However Motif repackages this shell so that dialogs become a subclass widget.

39.1.2 Constraint Widgets

Constraint widgets are concerned with the positioning and alignment of widgets contained within them. *XmManager* is a (Motif) subclass of the constraint widget specifying general manager facilities concerned with,for example, callbacks and highlight colour.

39.1.3 Construction widgets

Motif defines two basic classes of widgets that provide the basic building blocks of any GUI: **primitive** and **manager**. The majority of the remainder of this text is devoted to these widget classes.

Within each of these basic classes, several different sub-classes of widget are defined.

The broad function of each class is as follows:

Primitive widgets are designed to work as a single entity. They provide the building blocks with which we assemble our GUI. The PushButton is an example of a primitive widget class.

Manager widgets are designed to be *containers* and may have primitive and manager widgets placed under their control. The main function of manager widgets is to help control the design of a GUI. Manager widgets control how we organise a GUI by prescribing standard, or uniform, layouts (such as the MainWindow widget, Chapter 42) or providing widgets that let us place widgets in an ordered fashion (*e.g. RowColumn* or Form widgets, Chapter 41).

Following Chapters will give details and examples of all types of widgets. For the remainder of this chapter we will give a brief introduction to these widgets.

39.2 Primitive Widgets

The following primitive widgets are defined in Motif:

ArrowButton — A button with an orientatable arrow (Fig. 39.2). This button is defined and used in a similar fashion to the PushButton widget. An example of the ArrowButton can be seen in the `arrows.c` program in Chapter 41.

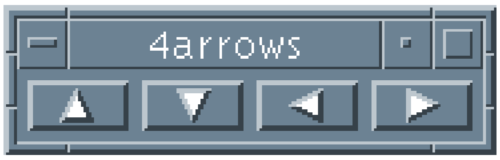


Figure 39.2: The Four Orientations of the ArrowButton Widget

Label — A widget which has text or an image (Pixmap) associated with it (Fig 39.3). The basic Label widget, as its name implies, does not do anything interactively. Its sole purpose is to help facilitate visual aids within the GUI. The Label widget does, however, have four (interactive) sub-classes of button:

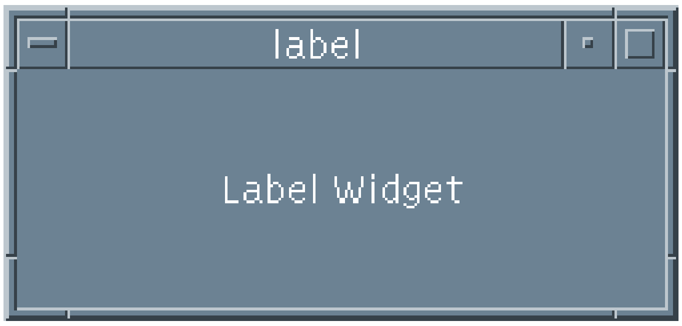


Figure 39.3: A Label Widget

PushButton — A button that can be labelled with a String (Fig 38.1). We have already met this widget in our first program, `push.c` (Chapter 38).

DrawnButton — A button with which an icon (Pixmap) can be associated (Fig. 39.4).

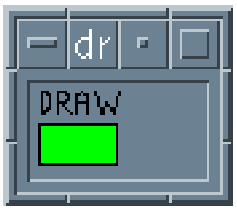


Figure 39.4: A DrawnButton Widget

CascadeButton — A button usually associated with a PullDown menu (Fig. 42.2). This widget is described in association with PullDown menu widgets in Chapter 42.

ToggleButton — A button that displays text or graphics together with a graphic indicator of the state of the ToggleButton. The ToggleButton has two states: either *on* or *off*. Toggle buttons may be grouped together to provide a variety of configurations. *RadioBox* are groups of ToggleButtons, where only one button can be selected at a time. A *CheckBox*, on the other hand, allows any number of buttons to be selected at a given time (Fig. 39.6). Examples of both configurations of ToggleButton are given in Chapter 48.

Scrollbar — A widget that allows the contents of a window to be displayed in a reduced area where the user can *scroll* the window to view hidden parts of the window. This widget can be defined and used explicitly in Motif programs (Chapter 47). Frequently, scrolling control will be defined when the ScrollBar (Fig. 39.7) is *compounded* with another widget. Chapters 44 and 50 discuss examples of the Scrollbar used in conjunction with Text (Fig. 39.10) and DrawingArea widgets respectively.

Separator — A widget used to separate items in a GUI to aid visual display. Fig. 39.8 illustrates the use of the *Separator* widget to delineate items (*Load* and *Quit*) in a single pulldown menu.

List — A widget that allows selection from a list of text items (Fig. 39.9). This widget is described in Chapter 45.

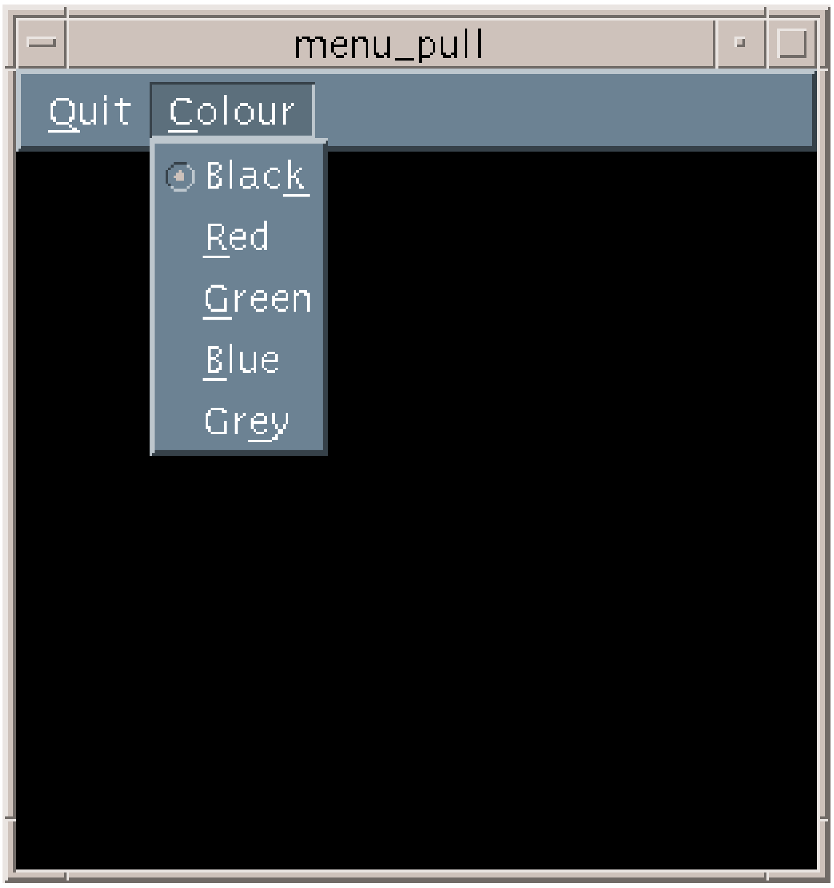


Figure 39.5: A CascadeButton Widget and Associated PullDown Menu

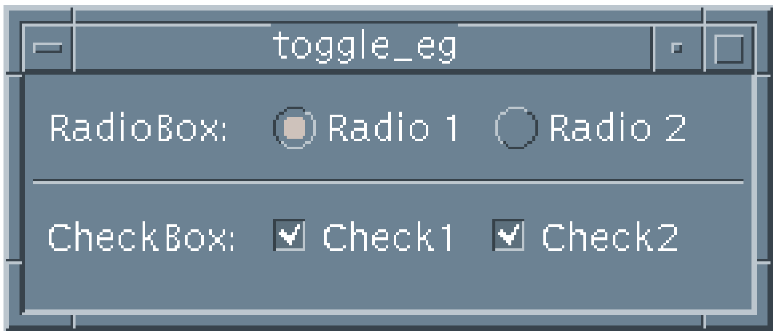


Figure 39.6: A RadioBox and CheckBox ToggleButton Widget Configuration

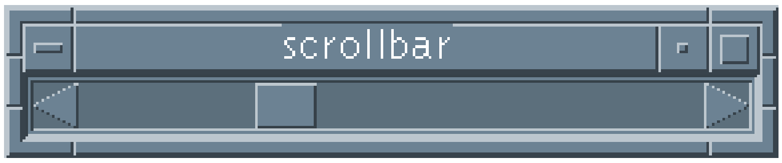


Figure 39.7: A Scrollbar Widget

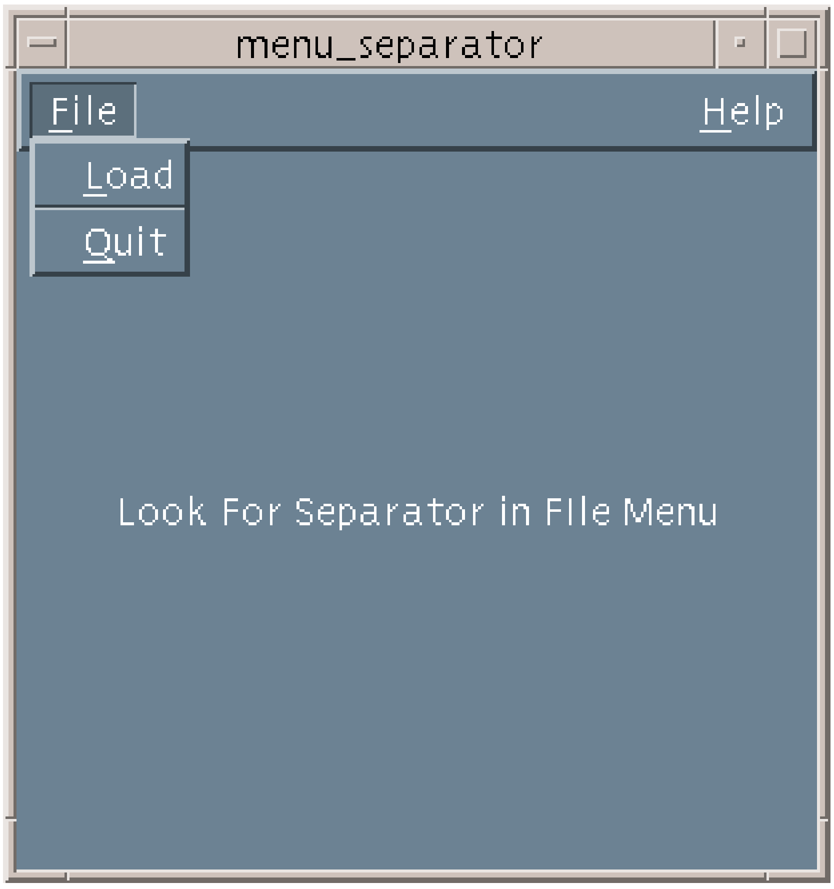


Figure 39.8: A Separator Widget Between Two Menu Items

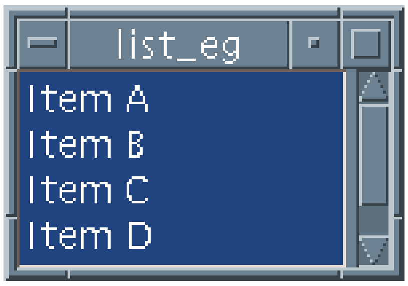


Figure 39.9: A List Widget

Text — A complete text editor widget (Fig. 39.10). This widget provides default callback resources for text selection, editing, cutting and pasting of text as well the editing style of the widget. Additional callback resources can easily be attached to extend and customise the text widget for many text editing applications. Chapter 44 addresses all such issues.

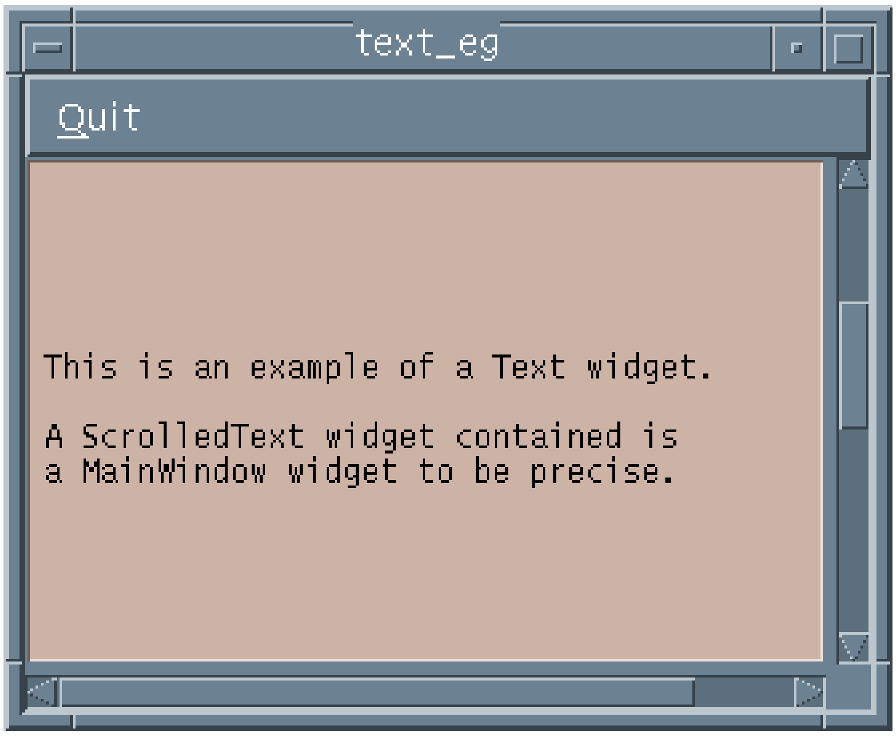
TextField — A single-line text editor. This is basically a Text widget which is limited to a single-line of text entry. The TextField widget has many uses in GUI design where simple input is required. For example, when a file name needs to specified as input in a GUI or a key word is required for some search.

Motif 2.0 also prescribes another form of text widget, **CSText**. This widget provides the same facilities a the Text widget but uses an alternative (compound) text string representation, `XmString` (Section 39.6), which is capable of supporting multiple fonts .

39.3 Gadgets

There may be occasions in Motif programming when we want to have the properties of a primitive widget but do not wish to worry about the management of the window properties of the widget. Motif has to manage each created widget's window and associated resources. If we have several widgets within our interface this could cause complications for our application.

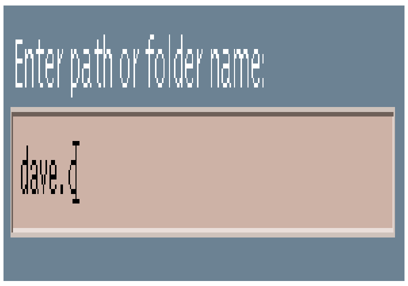
To attempt to alleviate many of these problems Motif provides *Gadgets*. Gadgets are basically windowless widgets and, therefore, require less



This is an example of a Text widget.

A ScrolledText widget contained is
a MainWindow widget to be precise.

Figure 39.10: A Text Widget



dave.c

Figure 39.11: A TextField Widget

resources than a widget. Control of the gadget is the responsibility of the parent of the gadget.

Not all widgets have corresponding gadgets. The following gadgets are available: **ArrowButton, Label** and **Separator**. They behave in a similar manner to their corresponding widgets.

We will not consider gadgets further in this book since our programs are relatively simple and the relevance of their use cannot be effectively illustrated. The primary use of gadgets is when there is a real need to save memory on the X server or within the application. Gadgets may actually increase computer processor load since X finds some events more difficult to track within them. Gadgets are basically an artefact from earlier versions of Motif that were developed when window construction and management was more critical. Later versions of X have optimised the X server and coupled with the fact that computer power has increased and memory is less expensive, the use of gadgets is not as important as it once used to be.

39.4 Manager Widgets

Manager widgets are the basic Motif widgets for constructing and organising our interfaces in Motif. The following *manager widget* classes are available:

Frame — A widget that provides an embossed effect to the *child* widget it contains (Fig 39.4). This is the simplest manager widget. An example of this widget is given in `frame.c` (*see* Exercise 39.1).

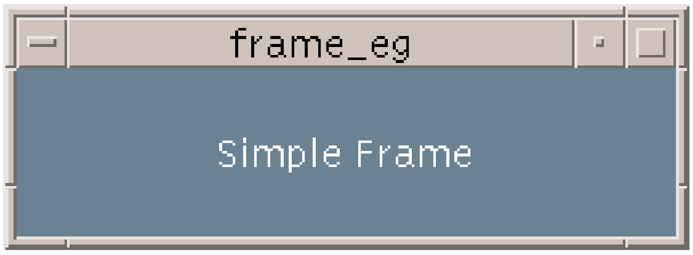


Figure 39.12: A Frame Widget

ScrolledWindow — A widget that allows scrolling of its child widget. Fig. 39.10 illustrates a common example of this: a *ScrolledText* widget that has a Text widget capable of being scrolled in a horizontal and vertical direction.

One common subclass of the ScrolledWindow widget is the MainWindow widget:

MainWindow — A typical *top level* container widget for an application. This widget provides a common, uniform *look and feel* for any GUI (similar to MS Windows *look and feel* (Section 36.1)). The MainWindow widget has well defined mechanisms for the provision of Menubars, Scrollbars and command and message windows (Fig. 39.13). Chapter 42 describes the major aspects of MainWindow programming.

DrawingArea — A widget where graphics can be displayed (Fig. 39.14). **Note:** Motif does not provide any graphics functions, Xlib provides all the graphic drawing and manipulation routines. Graphic programming and the interface with Xlib is one of the more difficult aspects of Motif to understand. Chapters 49—51 discuss these more advanced issues of programming and show how the DrawingArea widget is used in practice.

PanedWindow — Allows vertical tiling of child widgets. The use of this widget is not as common as other widgets and we will not address its use on this particular journey through Motif.

Scale — This widget provides a *slider* object that can be used for user input (Fig. 39.4). Chapter 46 describes the Scale widget.

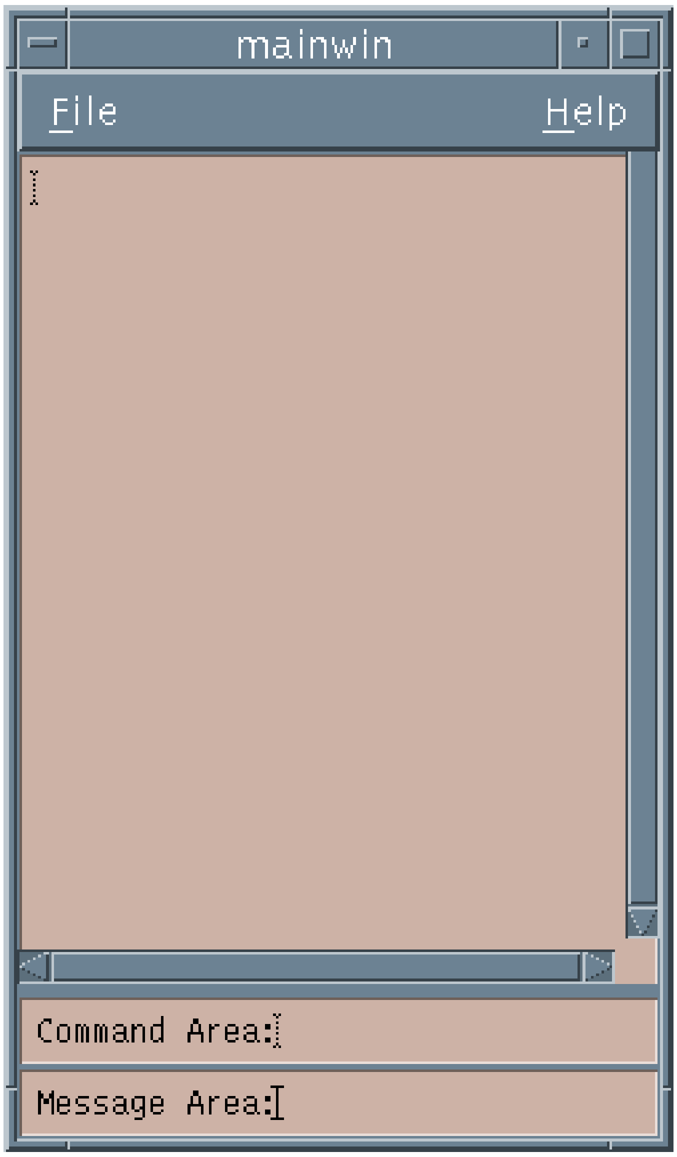
RowColumn — A widely used widget that can lay out widgets in an orderly 2D fashion. Chapter 41 describes this widget.

BulletinBoard — There are two sub-classes of this widget:

Form — A widget similar in use to *RowColumn* but allows greater control of the placement and sizing of widgets. Chapter 41 compares and contrasts the Form and RowColumn widgets.

Dialog — There are two forms of dialog:

- a *MessageBox* which simply gives information to the user (Fig. 39.16) and



Command Area:

Message Area:

Figure 39.13: A MainWindow Widget

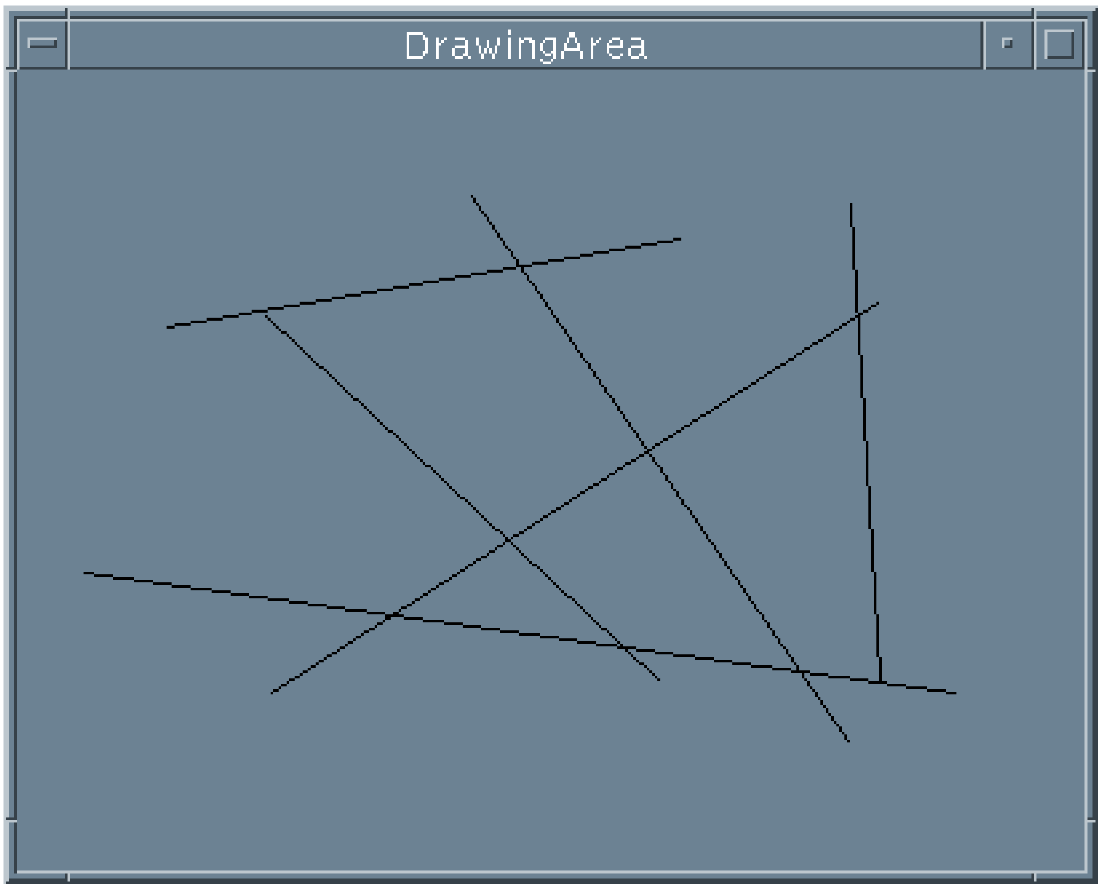


Figure 39.14: A DrawingArea Widget

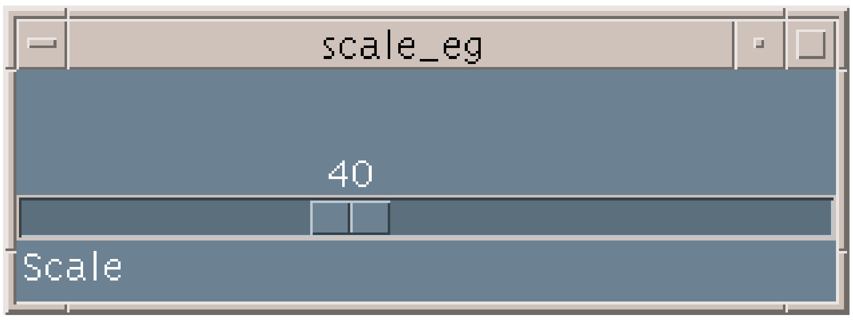
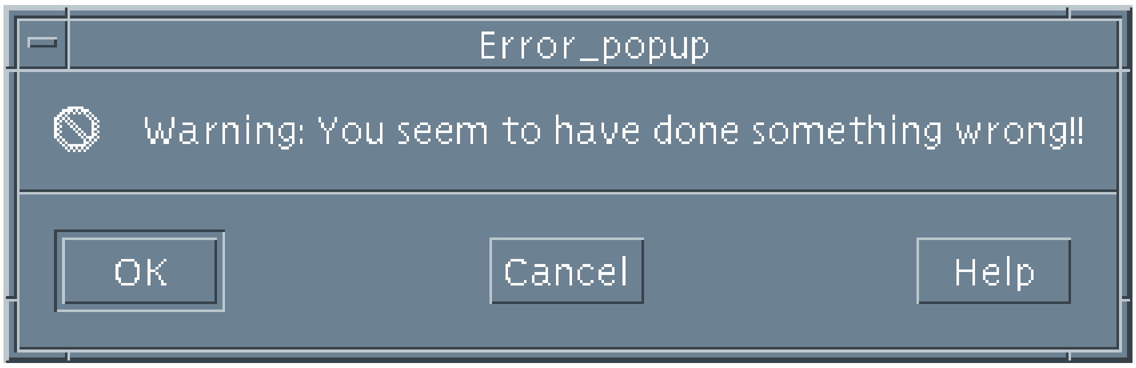


Figure 39.15: A Scale Widget

- a *SelectionBox* which allows interaction with the user. Motif provides two sub-classes of the *SelectionBox* widget: a *Command* widget for command line type input and a *FileSelectionBox* for directory/file selection (Fig. 39.17).



Warning: You seem to have done something wrong!!

Figure 39.16: A MessageBox (ErrorDialog) Widget

Dialog widgets, as the name implies, provide the direct line of communication between the user and the application. In the case of the *MessageBox* widgets this could simply be the program informing the user of some event or action. There are several prescribed classes of *MessageBox* widgets that are associated with a special event. For example there are *WarningDialog*, *InformationDialog* and *ErrorDialog* widgets (Fig. 39.16). Also some form of prescribed user interaction is provided by the *Command* and *FileSelectionBox* widgets. Motif also provides base *MessageBox* and *SelectionBox* widgets so that the programmer can assemble customised Dialogs. However, Motif programming style (Chapter 52) suggests that wherever appropriate the prescribed Dialog widgets should be used to provide uniformity across applications. Chapter 43 deals with many aspects of Dialog widget programming and usage.

39.4.1 Motif 2.0 Widgets

Motif 2.0 defines a few new Manager widgets:

ComboBox — A widget that combines the capabilities of a single line TextField and an XList.

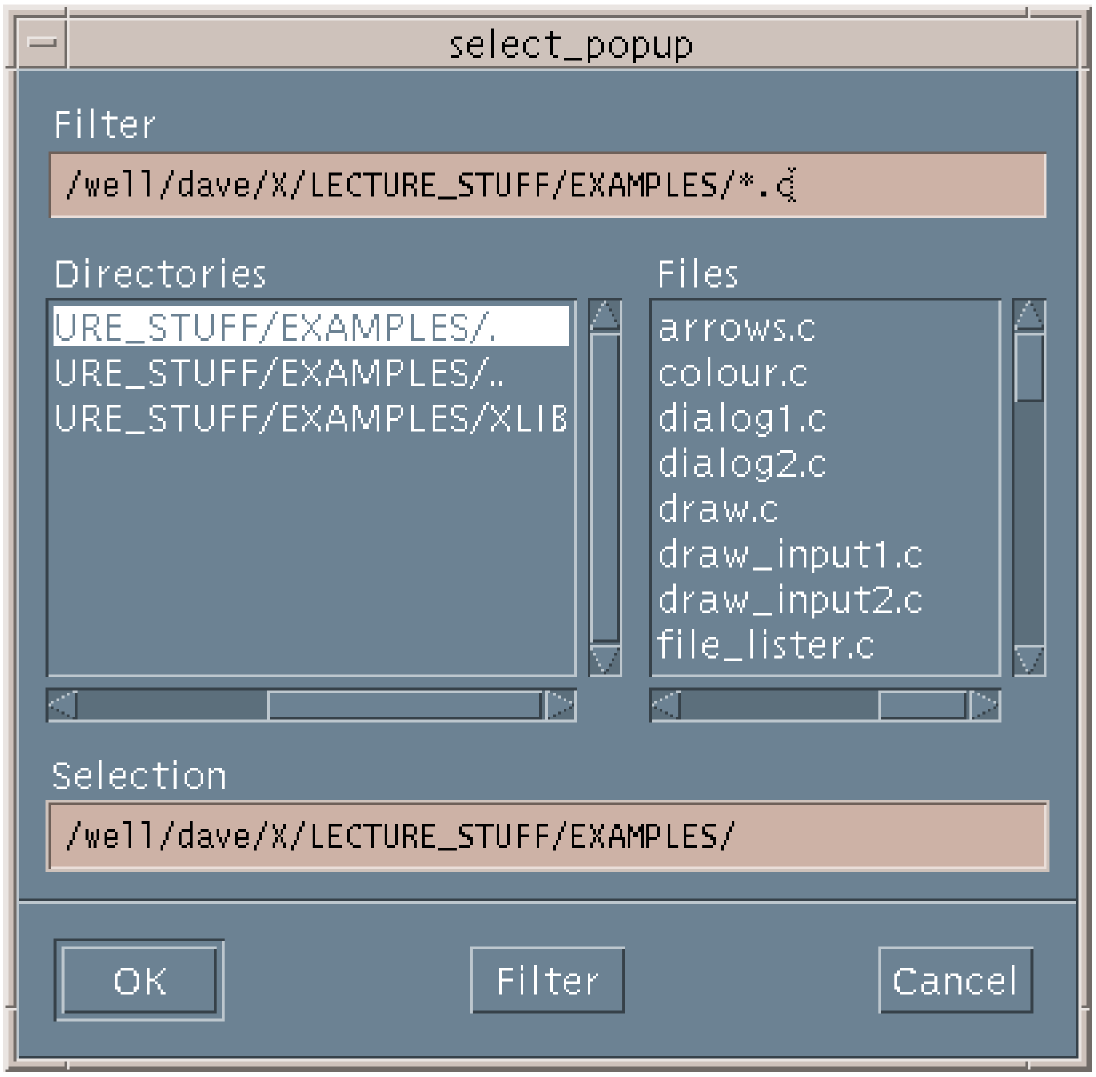


Figure 39.17: A FileSelectionBox Widget

IconGadget — A widget that can be used to display Icons.

Container — A widget that manages IconGadget children.

Notebook — A widget that organizes children into pages, tabs, status area and page scroller.

Scale (thermometer) — A modified version of the Scale widget with new resources added for thermometer behavior (see Chapter 46).

SpinBox — A widget that manages multiple traversable children.

The use of many of these new widgets is fairly advanced and, except where indicated, these widgets are not dealt with further in this introductory text.

39.5 Widget Resources

Each widget has a number of resources. These control many features of the widget such as the foreground and background colours, size *etc.*. A particular widget will have specialised resources such as callback resources which define how the widget responds to an event *etc.*.

Every widget is documented in the Motif Reference Manual which gives a complete list of the resources that a particular widget employs. When discussing individual widgets we will only consider the important resources that define the main characteristics of the widget concerned. The following Chapter addresses how widget resources can be set and altered for a given application.

There is a hierarchy of widgets (Fig 39.1) and a widget will *inherit* resources from a higher resource class in the widget hierarchy.

The levels of the hierarchy and related widget resources are:

Core — This *superclass* gives background, size and position resources that are common to all widgets.

XmPrimitive — The superclass of all primitive widgets defines resources related areas such as foreground and highlight colour.

Composite — The superclass of containers for widgets has two sub-classes:

Shell widgets have resources related to interfacing with the window manager.

Constraint widgets have resources that are concerned with the positioning and alignment of widgets contained within.

XmManager is a subclass of constraint and specifies general manager widget resources such as callbacks and highlight colour.

Widget level — Resources particular to a specific widget.

39.6 Strings in Motif

Motif programs (in C/C++) will typically need to use both types of *string* available:

- A data type `String` — A "normal" C string (an array of characters). However, for convenience Motif has defined `String` as a *standard* Motif Data Type. Clearly, as Motif programs are usually written in C, this data type will be the main means of communication between a program and the standard input and output mechanisms.

- A data type `XmString` — Motif *internal* string data structure. When Motif needs to *draw* strings on the display, to achieve this more information is needed than simply the array of characters. Consequently, Motif defines an `XmString` with a more complicated structure. Most widgets can have a label resource associated with them, which is usually an XmString data type. It is rarely that the Motif programmer will need to know the *internal* structure of an XmString.

Motif provides a number of functions to convert a `String` into an `XmString` or vice versa:

`XmStringCreateLocalized(),XmStringCreateLtoR(), XmStringGetLtoR(), XmStringCompare(), XmStringConcat(), XmStringCopy()`, *etc.* are common examples. Many behave in a similar manner to their C standard library string handling function counterparts.

Their use is fairly straightforward — the reference manuals should be consulted for more details.

39.7 Exercises

Exercise 39.1 *Run the following program (*`frame.c`*) and note the difference in appearance and interaction of the widget with the* `push.c` *program (Chapter 38). Note the effect of the* `XmNshadowType` *resource. What other settings are available for this resource and what effect do they have?*

```
/* frame.c --
 mount pushbutton of push.c in a frame widget
 */

#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Frame.h>  /* header file for frame stuff */

/* Prototype callback */

void pushed_fn(Widget , XtPointer ,
               XmPushButtonCallbackStruct *);

main(int argc, char **argv)
{
    Widget        top_wid, button, frame;
    XtAppContext app;


    top_wid = XtVaAppInitialize(&app, "Push", NULL, 0,
        &argc, argv, NULL, NULL);


    frame = XtVaCreateManagedWidget("frame",
         xmFrameWidgetClass, top_wid,
         XmNshadowType, XmSHADOW_IN,
         NULL);


    button = XmCreatePushButton(frame, "Push_me",
        NULL, 0);
    XtManageChild(button);

    XtAddCallback(button, XmNactivateCallback, pushed_fn, NULL);

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

void
pushed_fn(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)

{
    printf("Don't Push Me!!\n");
}
```

Exercise 39.2 *Rewrite the* `frame.c` *program (Exercise 39.1) so that it displays a button labelled "Quit Frame" which terminates the program when the button is depressed with the left mouse button.*

Exercise 39.3 *Write a program that inputs the string "X Convert Me!!" as a standard* `String` *data type and converts the string to an* `XmString` *data type. The program should also extract the substring "Convert Me!!" from the* `XmString` *and store it as a* `String`.

Chapter 40

Widget Resources

Every Motif widget has a number of resources that control or modify its behaviour and appearance. Depending on the widget's class, resources control aspects such as the size of the widget, the colour of the widget, whether scroll bars should be displayed and many other properties.

When a widget is created it inherits resources from a higher widget class and also creates its own (Section 39.5). All these resources have default values. It would be tedious to specify 30 plus resource values each time we create a widget.

However the application programmer or user may want to customise one or two resources in order to control a widget's size in a GUI or its dimensions on the screen, for example.

Throughout this Chapter we will use the PushButton program, `push.c`, developed in Chapter 38 as a case study and we shall see how we alter the size (width and height) of the button. The resource variables `XmNwidth` and `XmNheight` hold these values.

There are a few ways in which we can alter a particular resource's value.

40.1 Overriding Resource Defaults

Broadly, there are two methods of altering resources in Motif:

- External resource files — resource values are set and stored in particular files.

- Hard Coding in the program — resource values are set in the program code.

The advantage of using external files is that it allows the user to customise Motif applications without having to recompile the program. The user may never need to access the source code. Applications may also be customised each time they are run. Due to the wide variety of possible systems running X and a vast range of user needs this can be a useful feature. For example, screen resolutions vary a great deal from device to device and what may be a visually adequate size for a widget on one display may be totally inadequate on another. Also certain resources, particularly colour displays, may vary substantially across different platforms.

There are four basic ways to externally customise an application:

User resource file — a file placed in the users home directory called `.Xdefaults`.

Class resource file — a single application can have a special file reserved for its particular resource setting commands.

Command line parameters — a widget's resources can be specified when the program is actually run.

RESOURCE_MANAGER/ SCREEN_RESOURCES properties — A standard X Window program, xrdb, can be used to set certain application resources.

One problem with the external setting of resource values occurs if it has already been hard-coded, since hard-coded resource values have a higher precedence than all externally set resources.

There are some advantages to hard-coding resource values:

- It may be essential to prevent a user changing particular resource values. For example, changing certain widget sizes may totally disorganise the GUI design and display.
- Resource files require specific names and locations that may be hard to install and maintain correctly. File and directory permissions may occasionally be set so as to make the files unreadable. Files and directories sometimes get deleted or moved.

- Unix environment variables sometimes cause conflicts with external resource value settings.

A trade-off is sometimes required between applying hard-coding and allowing user freedom.

There is one other internal coding method for changing resources, by using what are called *fallback resources*. As their name implies fallback resources are set only if a resource has not been set by any of the above means. Fallbacks are therefore useful for setting alternatives to the default Motif resource settings.

The remaining sections in this Chapter detail how each individual resource setting method may be used.

40.2 User Resource (`.Xdefaults`) File

When Motif initialises the application it looks for the `.Xdefaults` file in your `HOME` directory. The `.Xdefaults` file is a standard text file, where each line may contain a resource value setting, a blank line or a comment denoted by an exclamation mark (!). This file contains directives that can reset any resource for any application, using the following method:

- The resources relating to widgets created within a particular application will be referred to by the application class name set when the `XtVaAppInitialize()` is called (Section 38.6.1). The application class name was set to "`Push`" in our example. Application names can also be used to refer to resources, but this is not as common as the class name. The application name is the name of the compiled program, `push` in this example.

- The resource name for a particular widget will have been specified in the program via the `name` argument of the widget creation function (Section 38.6.2). The name of the PushButton in our example is "`Push_me`". **Note:** this is not the variable name but the `name` argument in the create widget function.

- The resource value is then accessed via:

  application_class_name.widget_name.resource_name.

Note that when referring to resource names outside a program, the `XmN` part of the resource is dropped from the resource name. So, the width and height of the PushButton widget in the `push.c` application are referred to by:

`Push.Push_me.width` or `Push.Push_me.height`

- The directive in the file to set the width and height of the PushButton widget of `push.c` is of the form:

  ```
  ! Comment: Set push application, PushButton dimensions

  Push.Push_me.width: 200
  Push.Push_me.height: 300
  ```

  where 200 and 300 are the widget's new dimensions. Note that there is a colon separating the resource name and its new value.

  Wild card (∗) settings are also allowed. Therefore to set *all* widgets called "`Push_me`" width and height you could write:

  ```
  ! Comment: wildcard setting of widget Push_me dimensions

  *Push_me.width: 200
  *Push_me.height: 300
  ```

Motif widget class names may also be used. This will set *all* widgets of a given class and application to the same values so care should be taken. An alternative method to set the width and height of the PushButton of `push.c` is to use the Motif `XmPushButton` class name to set its resources via:

```
! Comment: Set push application,
! XmPushButton widget class dimensions

Push*XmPushButton.width: 200
Push*XmPushButton.height: 300
```

40.3 Class Resource File

The class resource file relates to a particular application, or class of applications. The application class name created with `XtVaAppInitialize()` is used to associate a class resource file with an application. Thus, the `push.c` program has an application class "`Push`" and therefore the application would have a class file named Push. The class resource files are normally stored in the user's home directory.

It is possible, and quite practical, to associate several applications with a single class name (by setting the appropriate `XtVaAppInitialize()` argument) and therefore with a single class resource file. This would allow for one class file to control the resource setting for many usually similar applications. Individual applications can again be referred to by their (compiled) program name, but this is not common.

The setting of individual resource values is as described for the `.Xdefaults` file, except that the wild card matching may not have such far reaching consequences. Therefore, to set the dimension of the PushButton in `push.c` we could write:

```
! Class Resource File for push.c called "Push"
! Store in HOME directory

*Push_me.width: 200
*Push_me.height: 300
```

40.4 Command Line Parameters

Resource values can be set with X window command line parameters. Some common resources can be easily referred to and a general command exists to set any resource value. The advantage of using the command line is that resource values can be altered each time the program is run. This is ideal if you only change a resource infrequently, or you are experimenting to find suitable resource values. However, setting resource values in this fashion regularly involves a lot of typing, so alternative methods should be considered.

Common resource values have special abbreviations for command line operation. These are listed in Table. 40.1

Option	**Data Passed**	**Use**
`-bg`	background colour	Sets background colour
`-fg`	foreground colour	Sets foreground colour
`-geometry`	`WidthxHeight+ Xorigin+Yorigin`	Sets window dimensions and location
`iconic`		Starts application as an icon
`-name`	Application name	Sets instance name **not** class
`-title`	Application title	Sets window title bar name
`-display`	Display name	Sets X server connection
`-xnllanguage`	language	Sets internationalisation locale
`-xrm`	X resource manager string	Set any other resources

Table 40.1: Command Line Resource Options

The foreground and background colours are referred to by the common colour name database which is detailed in the reference manuals (Section 51.4). So to set the background to `red` for the `push.c` program. We would run the program from the command line as follows:

```
push -bg red
```

The `-geometry` option sets the windows dimension and position. It has 4 parameters:

```
WidthxHeight+Xorigin+Yorigin
```

where `Width, Height, Xorigin` and `Yorigin` are integer values separated by `x` or `+`.

Therefore to set the window size to 300 by 300 with the origin at top left corner run the program from the command line with the following arguments:

```
push -geometry 300x300+0+0
```

You can omit either the size, `WidthxHeight`, or position, `+Xorigin+Yorigin` parameters in order to either size or position a window.

Therefore to set the window size to 500 by 500 and to allow the window manager to place it somewhere type:

```
push -geometry 500x500
```

and to explicitly position a window and use the default dimension type:

```
push -geometry +100+100
```

The `-xrm` option allows the user to set resources that are not otherwise facilitated from the command line. Following the `-xrm` option you supply a string that contains a resource setting similar to a single line resource value setting in a user or class resource file (including wildcards).

Therefore to change the highlight colour of the PushButton in `push.c` we could type:

```
push -xrm "Push*highlightColor:  red"
```

where `Push` is the application class name and `(XmN)highlightColor` is the resource being set to `red`.

Multiple settings of resource values require `-xrm` calls for each resource value setting. Therefore to set the highlight and foreground colours for the above PushButton we could type:

```
push -xrm "Push*highlightColor: red" \
          -xrm "Push*foreground: blue"
```

40.5 The Resource Manager Database

The X system provides a program, xrdb, that allow you to set up and edit resources stored on the root window of a display. The data is stored in `RESOURCE_MANAGER` property and, also, in `SCREEN_RESOURCES` property if multiple screens are supported. For the sake of simplicity we will assume that we are only working with a single screen and therefore do not need to consider the `SCREEN_RESOURCES` property further.

When xrdb is run it reads a `.Xresources` file from which it creates the `RESOURCE_MANAGER` property. The `.Xresources` file is usually stored in the users home directory. The `.Xresources` file can contain similar resource value settings as described for the `.Xdefaults` file and class resource file. However, xrdb can actually run the .Xresources or an other input file through the C preprocessor, so C preprocessor syntax is allowed in the .Xresources file. Several macros are defined so that constructs like `#ifdef` and `#include` may be used. One common example of their use is to set separate colour and monochrome resources for different screen settings. The macro `COLOR` is defined if a colour screen is present for a given display. Therefore, an .Xresources file could look for this and take appropriate actions:

```
#ifdef COLOR
! Colour screen detected set colour resources

Push*foreground: RoyalBlue
Push*background: LightSalmon

#else
! must be a monochrome screen

Push*foreground: black
Push*background: white
#endif
```

For further information on *xrdb* readers should consult the X Window system user reference manuals, or online manual documentation.

One advantage of this method of setting resources is that it allows for dynamic changing of defaults without editing files. In fact, the setting of resources via files may not work on X terminals with limited computing power, or when programs are run on multiple machines (since other machines may not have the appropriate `.Xdefaults` or class resource files). Having all the resource information in the server means that the information is available to all clients.

40.6 Hard-coding Resources Within a Program

There are two basic methods for setting resource values from within a C program. The first method described below is dynamic, meaning that resources can be set and altered at any occasion from within the program. Another method allows resources to be set when a widget is created. Both these methods would override other methods of resource setting.

40.6.1 Dynamic control of resources

Using this method we can change the values of a resource at any time from within a program. Typically two functions are employed to do this:

- We use `XtSetArg()` to place resource values in an `Arg`ument list.
- `XtSetValues()` then sets these values in the list for a given widget. We need to specify the widget whose resources we wish to set, the `arg` list and the number of arguments in the list as parameters to this function.

Therefore, to set the size of the `button` resources in `push.c` we would use `XtSetArg()` to set width and height values in the `arg` list and then set the `arg` values for the `button` widget as follows:

```
Arg args[2]; /* Arg array */
int n = 0; /* number arguments */

XtSetArg(args[n], XmNwidth, 500);
n++;
XtSetArg(args[n], XmNwidth, 750);
n++;
XtSetValues(button, args, n);
```

A related function `XtGetValues()` exists to find out resource values (Chapters 45 and 47 contain some examples of `XtGetValues()` in use).

A convenience function `XtVaSetValues()` actually combines the above two operations and make programming a little less tedious. XtVaSetValues sets the resource value pair for a given widget using a `NULL` terminated list much like `XtVaCreateManagedWidget()` (*see* below).

Therefore, we can achieve the same result as above for setting the `button` in `push.c` via:

```
XtSetValues(button, XmNwidth, 500, XmNwidth, 750, NULL);
```

40.6.2 Setting resources at creation

We can set resource values at creation. This is common if we wish to permanently override a default resource value. There are a couple of methods we can adopt to achieve this.

We can set an `Arg` argument list using `XtSetArg()` as in the previous section and then specify the `arg` list and the number of arguments in the `XmCreate...()` function. Therefore, we could amend the `push.c` program to set the resources at initialisation of the `button` widget by inserting the following code:

```
Arg args[2]; /* Arg array */
int n = 0; /* number arguments */

......

XtSetArg(args[n], XmNwidth, 400);
n++;
XtSetArg(args[n], XmNwidth, 600);
n++;

button = XmCreatePushButton(top_wid, "Push_me", args, n);
```

There is an alternative method which lets us set resource values and create a managed widget in one function call. The function is `XtVaCreateManagedWidget()` and is a convenient way to program such tasks. Note that this is a *general* function that can create many different classes of widget. This function is preferred for the creation of widgets due to its uniformity of syntax/structure and its brevity in performing more than one task. Where appropriate all widgets will subsequently be created using this function.

Let us look a the syntax of this function:

```
Widget XtVaCreateManagedWidget(String name,
               WidgetClass widget_class, Widget parent,
               ... resource name/value pairs ...,
               NULL)
```

where:

- `name` specifies the name for the created widget.
- `widget_class` specifies the class of the widget.
- `parent` is the parent widget.
- There then follows a `NULL` terminated list of pairs of resource name and values.

The function returns a widget of the class specified.

Therefore, to create a managed PushButton widget with dimensions 400 by 300 we would type:

```
button = XtVaCreateManagedWidget("Push_me",
  XmPushButtonWidgetClass, top_wid,
  XmNwidth, 400,
  XmNheight, 200,
  NULL);
```

Here, `XmPushButtonWidgetClass` is the class identifier of a PushButton. Other widget types should be fairly obvious.

40.7 Fallback Resources

Fallback resources are used as a mechanism where the specified resource value settings only take effect if all other resource setting methods have failed. Fallback resources are passed as arguments to the `XtVaAppInitialize()` function (Section 38.6.1). The fallback resources are passed as a `NULL` terminated list of `String`s to this function. Each `String` specifies a resource value setting similar to those developed for the user and class resource files. Therefore, to set fallback resources for the `push.c` program we could include the following code:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

/* Define fallback\_resources  */

static String fallback\_resources[] = {
       "*width: 300",
       "*heigth: 400",
       NULL /* NULL termination}
       };

main(int argc, char **argv)
{
    Widget        top_wid, button;
    XtAppContext app;

    top_wid = XtVaAppInitialize(&app,
                  "Push", /* class name */
                   NULL, 0, /* NO command line options table */
                   &argc, argv,  /* command line arguments */
                   fallback\_resources, /* fallback\_resources list */
                   NULL);


     .........
```

Chapter 41

Combining Widgets

The programs we have studied so far have used a top level (application) shell widget and a single child widget (*e.g.* PushButton).

Most Motif programs will need to employ a combination of many widgets, in order to produce an effective GUI. For example, a text editor application usually requires a combination of buttons, menus, text areas *etc.*

In this Chapter we will look at how we arrange widgets and furthermore explicitly position widgets within a GUI.

41.1 Arranging and Positioning Widgets

We should now be familiar with the concept of a *tree of widgets* which is formed by creating widgets with other widgets as parents. When we combine widgets, we simply carry this principle further. Our major concern when combing widgets is to place them in some order and some relative position, with respect to other widgets. Usually we do not want widgets to obscure each other. Care must also be taken with the organisation and positioning of widgets when the window containing the widgets is resized. In particular:

- Is it practical for the particular widget to be resized?
- Can the relative positioning between widgets be preserved as a result of the resize?

As we shall discover shortly, Motif provides a great deal of flexibility in the behaviour of widgets in such circumstances. The GUI programmer should carefully consider the means of interaction most suitable for the particular application and then select the most appropriate Motif widget and means of organisation to achieve this.

The management of widget geometry is taken care of by certain **manager widgets**. The *RowColumn* and *Form* widgets are the most common widgets used for arranging widgets.

Consider a simple multiple widget program output (Fig 41.1).

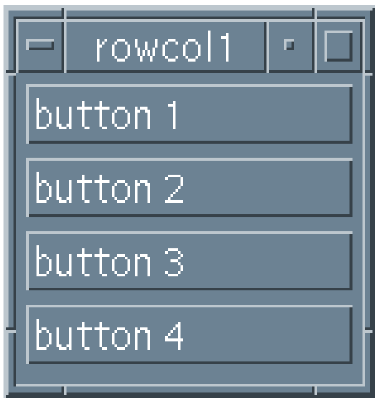


Figure 41.1: Simple Multiple Widget Layout

This layout is usually specified by the widget tree structure illustrated in Figure 41.2.

The application shell is still be the top level, below this there would be a RowColumn or Form widget which would contain some primitive widgets (*e.g.* PushButtons) and define exactly how they are to be positioned relative to each other. Several possible arrangements are available and the format of each depends on the context of the application (*e.g.* how the widgets are displayed if the window size is increased or decreased).

We will now consider how we can arrange widgets by considering the RowColumn and Form widgets.

41.2 The RowColumn Widget

This the simplest widget in terms of how it manages the positioning of its child widgets. Widgets are positioned as follows:

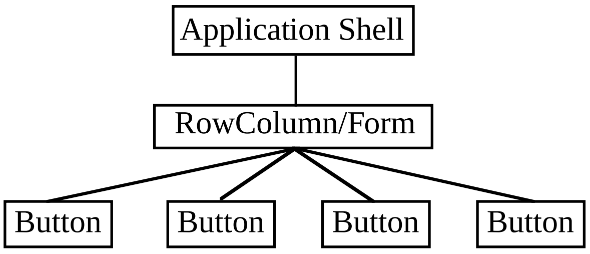


Figure 41.2: Multiple Widget Tree

- Consecutively created child widgets are layed out in a horizontal or vertical order depending on how the `XmNorientation` resource is set. `XmVERTICAL` is the default value, `XmHORIZONTAL` is the alternative value.
- To specify the number of rows/columns, set the `XmNnumcolumns` resource. This sets the number of columns if the `XmNorientation` is `XmVERTICAL`, otherwise it specifies the number of rows.
- Widgets must be the same size, otherwise the RowColumn widget will force widgets to be the same size. The resizing of widgets is controlled by the `XmNpacking` resource:
  - If the packing is set to `PACK_TIGHT` (default), then columns (rows if `XmHORIZONTAL` orientation) are forced to have the same width.
  - `PACK_COLUMN` makes all children the same size.
  - `PACK_NONE` disables any attempt to make the children regular in size.

Let us now look at two programs that illustrate the above principles by studying how we achieve the outputs illustrated in Fig. 41.1 (`rowcol1.c` program) and Fig. 41.3 (`rowcol2.c` program). As can be seen in these figures, `rowcol1.c` lays 4 PushButtons vertically (default) whereas `rowcol2.c` sets the `XmNorientation` resource for a horizontal layout of the same 4 buttons. The output of the programs are illustrated in Fig 41.3.

The `rowcol1.c` program is as follows:

```
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
```

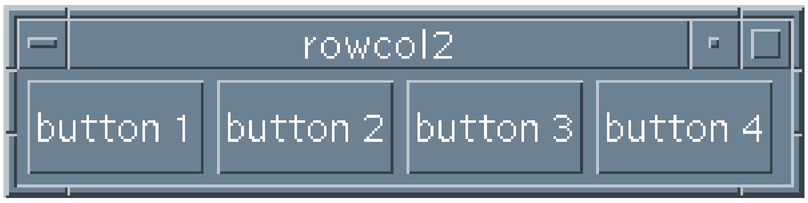


Figure 41.3: The `rowcol2.c` program output

```
main(int argc, char **argv)

{
    Widget top_widget, rowcol;
    XtAppContext app;

    top_widget = XtVaAppInitialize(&app, "rowcol", NULL, 0,
        &argc, argv, NULL, NULL);

    rowcol = XtVaCreateManagedWidget("rowcolumn",
        xmRowColumnWidgetClass, top_widget, NULL);

    (void) XtVaCreateManagedWidget("button 1",
        xmPushButtonWidgetClass, rowcol, NULL);

    (void) XtVaCreateManagedWidget("button 2",
        xmPushButtonWidgetClass, rowcol, NULL);

    (void) XtVaCreateManagedWidget("button 3",
        xmPushButtonWidgetClass, rowcol, NULL);

    (void) XtVaCreateManagedWidget("button 4",
        xmPushButtonWidgetClass, rowcol, NULL);

    XtRealizeWidget(top_widget);
    XtAppMainLoop(app);
}
```

The `rowcol1.c` program does not really do much. It provides no callback functions to perform any tasks. It simply creates a RowColumn widget, `rowcol`, and creates 4 child buttons with this. Note that `rowcol` is a child of `app` — the application shell widget. The `<Xm/RowColumn.h>` header file must also be included.

The `rowcol2.c` program is identical to `rowcol1.c`, except that the `XmNorientation` resource is set at the `rowcol` widget creation with:

```
rowcol = XtVaCreateManagedWidget("rowcolumn",
                xmRowColumnWidgetClass, top_widget,
                XmNorientation, XmHORIZONTAL,
                NULL);
```

41.3 Forms

Forms are the other primary geometry manager widget. They allow more complex handling of positioning of child widgets and can handle widgets of different sizes.

There is more than one way to arrange widgets within a form. We will look at three programs `form1.c, form2.c` and `form3.c` that achieve similar results but illustrate different approaches to attaching widgets to forms. All three programs produce initial output illustrated in Fig. 41.4 however if the window is resized, the different attachment policies have a different effect (Figs 41.5, 41.6 and 41.7).

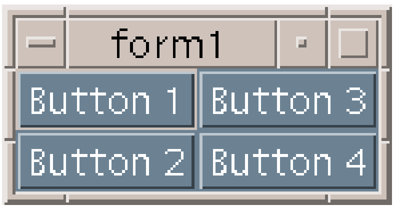


Figure 41.4: form1.c initial output

41.3.1 Simple Attachment —`form1.c`

Widgets are placed in a form by specifying the attachment of widgets to *edges* of other widgets. The edges of widgets can either be on the form widget itself, or on other child widgets. Edges are referred to by top, bottom, left and right attachments. A widget has resources, such as `XmNtopattachment` to attach a widget, to an appropriate edge:

- To attach a widget to the parent form, set the resource value `XmATTACH_FORM`.
- To attach a widget to another widget, set the resource value `XmATTACH_WIDGET`.
- The widget that an attachment is made to must also be specified, by setting the resource `XmNtopWidget` to the appropriate widget.

Let us look at how all this works in the `form1.c` program:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>


main (int argc, char **argv)

{ XtAppContext app;
  Widget   top_wid, form,
          button1, button2,
           button3, button4;
  int n=0;


  top_wid = XtVaAppInitialize(&app, "Form1",
        NULL, 0, &argc, argv, NULL, NULL);

  /* create form and child buttons */

  form = XtVaCreateManagedWidget("form",
        xmFormWidgetClass, top_wid, NULL);

  button1 = XtVaCreateManagedWidget("Button 1",
      xmPushButtonWidgetClass, form,
      /* attach to top, left of form */
      XmNtopAttachment, XmATTACH_FORM,
      XmNleftAttachment, XmATTACH_FORM,
      NULL);

  button2 = XtVaCreateManagedWidget("Button 2",
      xmPushButtonWidgetClass, form,
      XmNtopAttachment, XmATTACH_WIDGET,
      XmNtopWidget, button1, /* top to button 1 */
      XmNleftAttachment, XmATTACH_FORM, /* left, bottom to form */
      XmNbottomAttachment, XmATTACH_FORM,
      NULL);

  button3 = XtVaCreateManagedWidget("Button 3",
      xmPushButtonWidgetClass, form,
      XmNtopAttachment, XmATTACH_FORM, /* top, right to form */
      XmNrightAttachment, XmATTACH_FORM,
      XmNleftAttachment, XmATTACH_WIDGET, /* left to button 1 */
      XmNleftWidget, button1,
      NULL);

  button4 = XtVaCreateManagedWidget("Button 4",
      xmPushButtonWidgetClass, form,
      XmNbottomAttachment, XmATTACH_FORM, /* bottom right to form */
      XmNrightAttachment, XmATTACH_FORM,
      XmNtopAttachment, XmATTACH_WIDGET,
      XmNtopWidget, button3, /* top to button 3 */
      XmNleftAttachment, XmATTACH_WIDGET,
      XmNleftWidget, button2, /* left to button 2 */
      NULL);
  XtRealizeWidget (top_wid);
  XtAppMainLoop (app);
}
```

In the above program, the `form` widget is created as a child of `app` and 4 buttons are the children of `form`. The inclusion of `<Xm/Form.h>` header file is always required.

The attachment of the button widgets is as follows:

- `button1` is simply attached to the top left of the `form`.

- `button2` is attached to bottom left of the `form` and its top side is attached to `button1`.
- `button3` is attached to the top right of the `form` and its right side is attached to `button1`.
- `button4` is attached to the bottom right of the `form` and its top and left sides are attached to `button3` and `button2` respectively.

It is advisable to control the attachment as much as possible, so that any resizing of the form will still preserve the desired order. Fig 41.5 shows the effect of an enlargement of the window produce by the `form1.c` program. Notice that the relative sizes of individual buttons is not preserved.

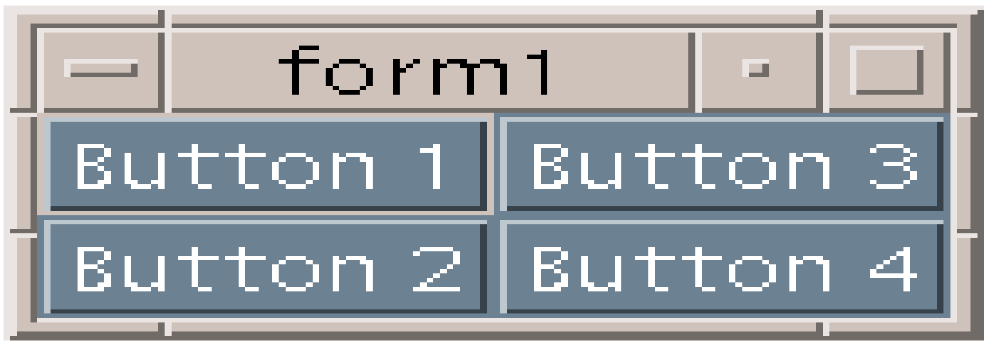


Figure 41.5: form1.c resized window output

41.3.2 Attach positions — `form2.c`

We can position the side of a widget to a *position* in a form. Motif assumes that a form has been partitioned into a number of segments. The position specified is the number of segments from the top left corner.

By default there is assumed to be 100 divisions along the top, bottom, left and right sides. This can be changed by setting the `form` resource `XmNfractionBase`

The position of a particular side of a widget is set by the widget resource `XmNtopAttachment` *etc.* to `XmATTACH_POSITION` and then setting another resource `XmNtopPosition` *etc.* to the appropriate integer value.

The `form2.c` program specifies the top left of `button1` to the edges of the `form`. The right and bottom edges are attached half way (50 out of a

100 units) along the respective bottom and right edges of the *form*. The `button2, button3` and `button4` widgets are positioned similarly.

The complete `form2.c` program listing is:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>


main (int argc, char **argv)

{ XtAppContext app;
  Widget   top_wid, form,
          button1, button2,
           button3, button4;
  int n=0;


  top_wid = XtVaAppInitialize(&app, "Form2",
        NULL, 0, &argc, argv, NULL, NULL);

  /* create form and child buttons */

  form = XtVaCreateManagedWidget("form", xmFormWidgetClass,
      top_wid, NULL);

  button1 = XtVaCreateManagedWidget("Button 1",
      xmPushButtonWidgetClass, form,
      /* attach to top, left of form */
      XmNtopAttachment, XmATTACH_FORM,
      XmNleftAttachment, XmATTACH_FORM,
      XmNrightAttachment, XmATTACH_POSITION,
      XmNrightPosition, 50,
      XmNbottomAttachment, XmATTACH_POSITION,
      XmNbottomPosition, 50,
      NULL);
  button2 = XtVaCreateManagedWidget("Button 2",
      xmPushButtonWidgetClass, form,
      XmNbottomAttachment, XmATTACH_FORM,
      XmNleftAttachment, XmATTACH_FORM,
      XmNrightAttachment, XmATTACH_POSITION,
      XmNrightPosition, 50,
      XmNtopAttachment, XmATTACH_POSITION,
      XmNtopPosition, 50,
      NULL);

  button3 = XtVaCreateManagedWidget("Button 3",
      xmPushButtonWidgetClass, form,
      XmNtopAttachment, XmATTACH_FORM,
      XmNrightAttachment, XmATTACH_FORM,
      XmNleftAttachment, XmATTACH_POSITION,
      XmNleftPosition, 50,
      XmNbottomAttachment, XmATTACH_POSITION,
      XmNbottomPosition, 50,
      NULL);


  button4 = XtVaCreateManagedWidget("Button 4",
      xmPushButtonWidgetClass, form,
      XmNbottomAttachment, XmATTACH_FORM,
      XmNrightAttachment, XmATTACH_FORM,
      XmNleftAttachment, XmATTACH_POSITION,
      XmNleftPosition, 50,
      XmNtopAttachment, XmATTACH_POSITION,
      XmNtopPosition, 50,
      NULL);


  XtRealizeWidget (top_wid);
  XtAppMainLoop (app);
}
```

One effect of this type of widget attachment is that the relative size of component widgets is preserved when the window containing these widgets is resized (Fig 41.6).

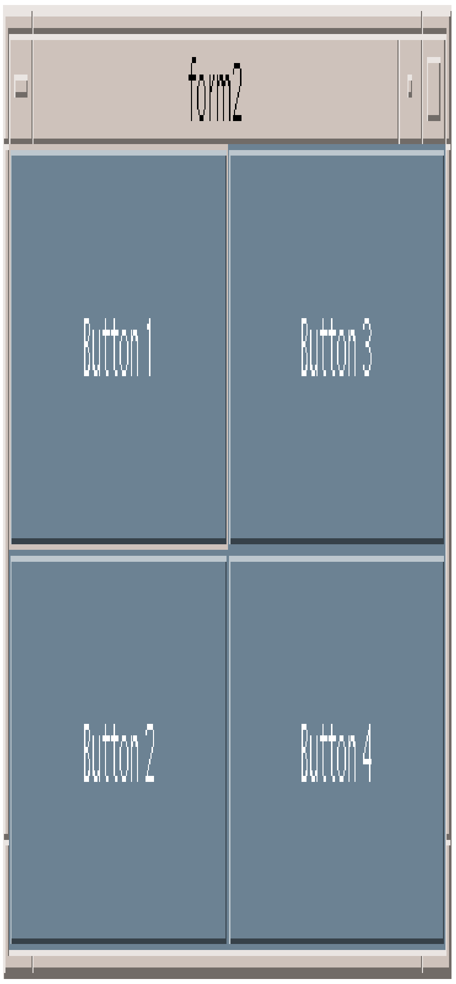


Figure 41.6: form2.c output (resized)

41.3.3 Opposite attachment — `form3.c`

Yet another way to attach widgets is to place an edge opposite edges of another widget. This is achieved by setting the `XmNtopAttachment` *etc.* resources to `XmATTACH_OPPOSITE_WIDGET`. Just as with `XmATTACH_WIDGET`, a widget has to be associated with the `XmNwidget` resource.

With the opposite form of attachment “similar” edges are attached. In the `form3.c` program we attach the right edge of `button2` to the right edge of `button1`, the left edge of `button3` with the left of `button1` and so on.

The complete `form3.c` program listing is:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>


main (int argc, char **argv)

{ XtAppContext app;
  Widget   top_wid, form,
          button1, button2,
           button3, button4;
  int n=0;


  top_wid = XtVaAppInitialize(&app, "Form3",
        NULL, 0, &argc, argv, NULL, NULL);

  /* create form and child buttons */

  form = XtVaCreateManagedWidget("form", xmFormWidgetClass,
      top_wid, NULL);

  button1 = XtVaCreateManagedWidget("Button 1",
      xmPushButtonWidgetClass, form,
      /* attach to top, left of form */
      XmNtopAttachment, XmATTACH_FORM,
      XmNleftAttachment, XmATTACH_FORM,
      NULL);


  button2 = XtVaCreateManagedWidget("Button 2",
      xmPushButtonWidgetClass, form,
      XmNtopAttachment, XmATTACH_WIDGET,
      XmNtopWidget, button1,
      XmNleftAttachment, XmATTACH_FORM,
      XmNbottomAttachment, XmATTACH_FORM,
      XmNrightAttachment, XmATTACH_OPPOSITE_WIDGET,
      XmNrightWidget, button1,
      NULL);

  button3 = XtVaCreateManagedWidget("Button 3",
      xmPushButtonWidgetClass, form,
      XmNtopAttachment, XmATTACH_FORM,
      XmNrightAttachment, XmATTACH_FORM,
      XmNleftAttachment, XmATTACH_WIDGET,
      XmNleftWidget, button1,
      NULL);


  button4 = XtVaCreateManagedWidget("Button 4",
      xmPushButtonWidgetClass, form,
      XmNbottomAttachment, XmATTACH_FORM,
      XmNrightAttachment, XmATTACH_FORM,
      XmNtopAttachment, XmATTACH_WIDGET,
      XmNtopWidget, button3,
      XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET,
      XmNleftWidget, button3,
      NULL);


  XtRealizeWidget (top_wid);
  XtAppMainLoop (app);
}
```

The relative sizing of widgets within the window is not guaranteed to be preserved, as in `form1.c` (Fig 41.7). However this method of layout is sometimes a natural way to express the configuration of the widgets.

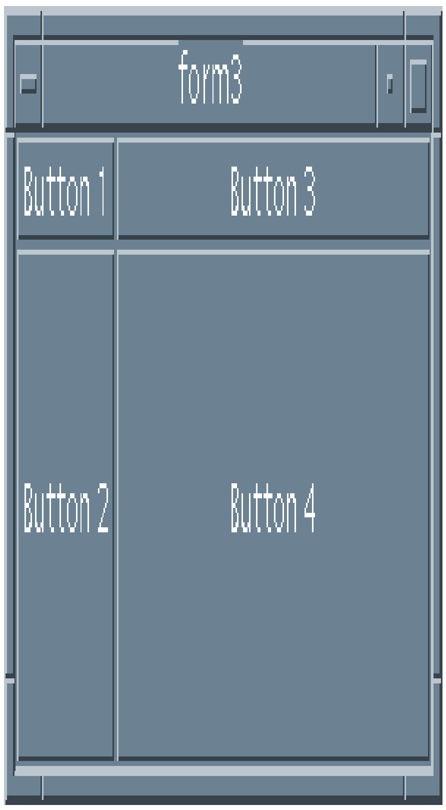


Figure 41.7: form3.c output (resized)

41.3.4 A more complete form program — `arrows.c`

Let us finish off this section by building a Form widget that contains two different types of widget. It will also use callback functions thus making a more complete application program example.

The program is called `arrows.c`. It creates 4 ArrowButton widgets arranged in a north, south, east and west type arrangement. In the middle of the 4 ArrowButtons is a PushButton, labelled "Quit". The output of `arrows.c` is shown in Fig. 41.8.

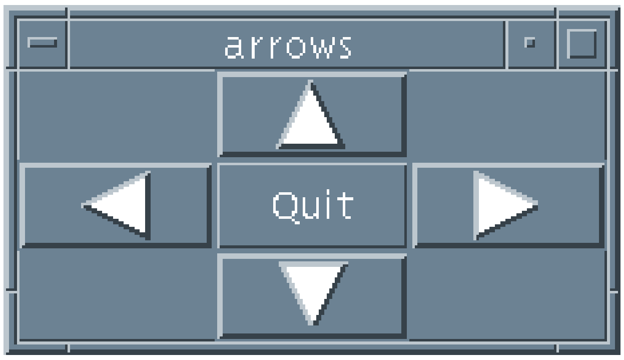


Figure 41.8: arrows.c output

The program listing is:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/ArrowB.h>
#include <Xm/Form.h>

/* Prototype callback fns */
void north(Widget , XtPointer ,
          XmPushButtonCallbackStruct *),
     south(Widget , XtPointer ,
          XmPushButtonCallbackStruct *),
     east(Widget , XtPointer ,
          XmPushButtonCallbackStruct *),
     west(Widget , XtPointer ,
          XmPushButtonCallbackStruct *),
     quitb(Widget , XtPointer ,
          XmPushButtonCallbackStruct *);


main(int argc, char **argv)

{
    XtAppContext app;
    Widget top_wid, form,
           arrow1, arrow2,
           arrow3, arrow4,
           quit;

    top_wid = XtVaAppInitialize(&app, "Multi Widgets",
                 NULL, 0, &argc, argv, NULL, NULL);


    form = XtVaCreateWidget("form", xmFormWidgetClass, top_wid,
                 XmNfractionBase,     3,
                NULL);

    arrow1 = XtVaCreateManagedWidget("arrow1",
                xmArrowButtonWidgetClass, form,
                XmNtopAttachment,    XmATTACH_POSITION,
                XmNtopPosition,      0,
                XmNbottomAttachment, XmATTACH_POSITION,
                XmNbottomPosition,   1,
                XmNleftAttachment,   XmATTACH_POSITION,
                XmNleftPosition,     1,
                XmNrightAttachment,   XmATTACH_POSITION,
                XmNrightPosition,    2,
                XmNarrowDirection,   XmARROW_UP,
                NULL);

    arrow2 = XtVaCreateManagedWidget("arrow2",
                xmArrowButtonWidgetClass, form,
                XmNtopAttachment,    XmATTACH_POSITION,
                XmNtopPosition,      1,
                XmNbottomAttachment, XmATTACH_POSITION,
                XmNbottomPosition,   2,
                XmNleftAttachment,   XmATTACH_POSITION,
                XmNleftPosition,     0,
                XmNrightAttachment,   XmATTACH_POSITION,
                XmNrightPosition,    1,
                XmNarrowDirection,   XmARROW_LEFT,
                NULL);

    arrow3 = XtVaCreateManagedWidget("arrow3",
                xmArrowButtonWidgetClass, form,
                XmNtopAttachment,    XmATTACH_POSITION,
                XmNtopPosition,      1,
                XmNbottomAttachment, XmATTACH_POSITION,
                XmNbottomPosition,   2,
                XmNleftAttachment,   XmATTACH_POSITION,
                XmNleftPosition,     2,
                XmNrightAttachment,   XmATTACH_POSITION,
                XmNrightPosition,    3,
                XmNarrowDirection,   XmARROW_RIGHT,
                NULL);

   arrow4 = XtVaCreateManagedWidget("arrow4",
                xmArrowButtonWidgetClass, form,
                XmNtopAttachment,    XmATTACH_POSITION,
                XmNtopPosition,      2,
                XmNbottomAttachment, XmATTACH_POSITION,
                XmNbottomPosition,   3,
                XmNleftAttachment,   XmATTACH_POSITION,
                XmNleftPosition,     1,
                XmNrightAttachment,   XmATTACH_POSITION,
                XmNrightPosition,    2,
                XmNarrowDirection,   XmARROW_DOWN,
                NULL);

   quit =XtVaCreateManagedWidget("Quit",
                xmPushButtonWidgetClass, form,
                XmNtopAttachment,    XmATTACH_POSITION,
                XmNtopPosition,      1,
                XmNbottomAttachment, XmATTACH_POSITION,
                XmNbottomPosition,   2,
                XmNleftAttachment,   XmATTACH_POSITION,
                XmNleftPosition,     1,
                XmNrightAttachment,   XmATTACH_POSITION,
                XmNrightPosition,    2,
                NULL);

    /* add callback functions */

    XtAddCallback(arrow1, XmNactivateCallback, north, NULL);
    XtAddCallback(arrow2, XmNactivateCallback, west, NULL);
    XtAddCallback(arrow3, XmNactivateCallback, east, NULL);
    XtAddCallback(arrow4, XmNactivateCallback, south, NULL);
    XtAddCallback(quit, XmNactivateCallback, quitb, NULL);

    XtManageChild(form);
    XtRealizeWidget(top_wid);
    XtAppMainLoop(app); }


    /* CALLBACKS */

void north(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)

    { printf("Going North\n");
    }

void west(Widget w, XtPointer client_data,
         XmPushButtonCallbackStruct *cbs)

    { printf("Going West\n");
    }

void east(Widget w, XtPointer client_data,
         XmPushButtonCallbackStruct *cbs)
    { printf("Going East\n");
    }

\begin{verbatim}
void south(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)

    { printf("Going South\n");
    }

void quitb(Widget w, XtPointer client_data,
          XmPushButtonCallbackStruct *cbs)

    { printf("quit button pressed\n");
      exit(0);
    }
```

The `arrows.c` program uses the *fraction base* positioning method of placing widgets within a form:

- The `XmNfractionBase` is reset to 3 thus creating a 3 by 3 grid for attaching widget edges.
- Each top, bottom, left and right edge of the 4 ArrowButtons and the “Quit” PushButton are attached to the appropriate position within the grid.
- Callback functions for each widget are added in the usual way. The 4 arrow widget callback functions simply print out their direction. The quit callback function exits the program.

Chapter 42

The MainWindow Widget and Menus

When designing a GUI, care must be taken in the presentation of the application's primary window. This window is important as it is likely to be the major focus of the application — the most visible and most used window in the application. In order to facilitate consistency amongst many different applications, Motif provides a general framework: the *MainWindow Widget*, for an application developer to work with. The *Motif Style Guide* (Chapter 52) recommends that, whenever applicable, the MainWindow window should be used. It should be noted, however, that the general framework of the MainWindow is not always applicable to every application front end GUI. A text editor application front end is an example that might easily map into the MainWindow framework, a simple calculator application most certainly would *not* fit the prescribed framework.

A MainWindow widget can manage up to five specialised child widgets (Fig 42.1):

- The *menu bar* — to which a number of pull down menus can be attached.
- The *work area* — the primary widget is placed here to perform the applications main functions.
- Vertical and Horizontal *scroll bars* — for the work area.
- The *command area* — where single-line commands can be entered by the user.

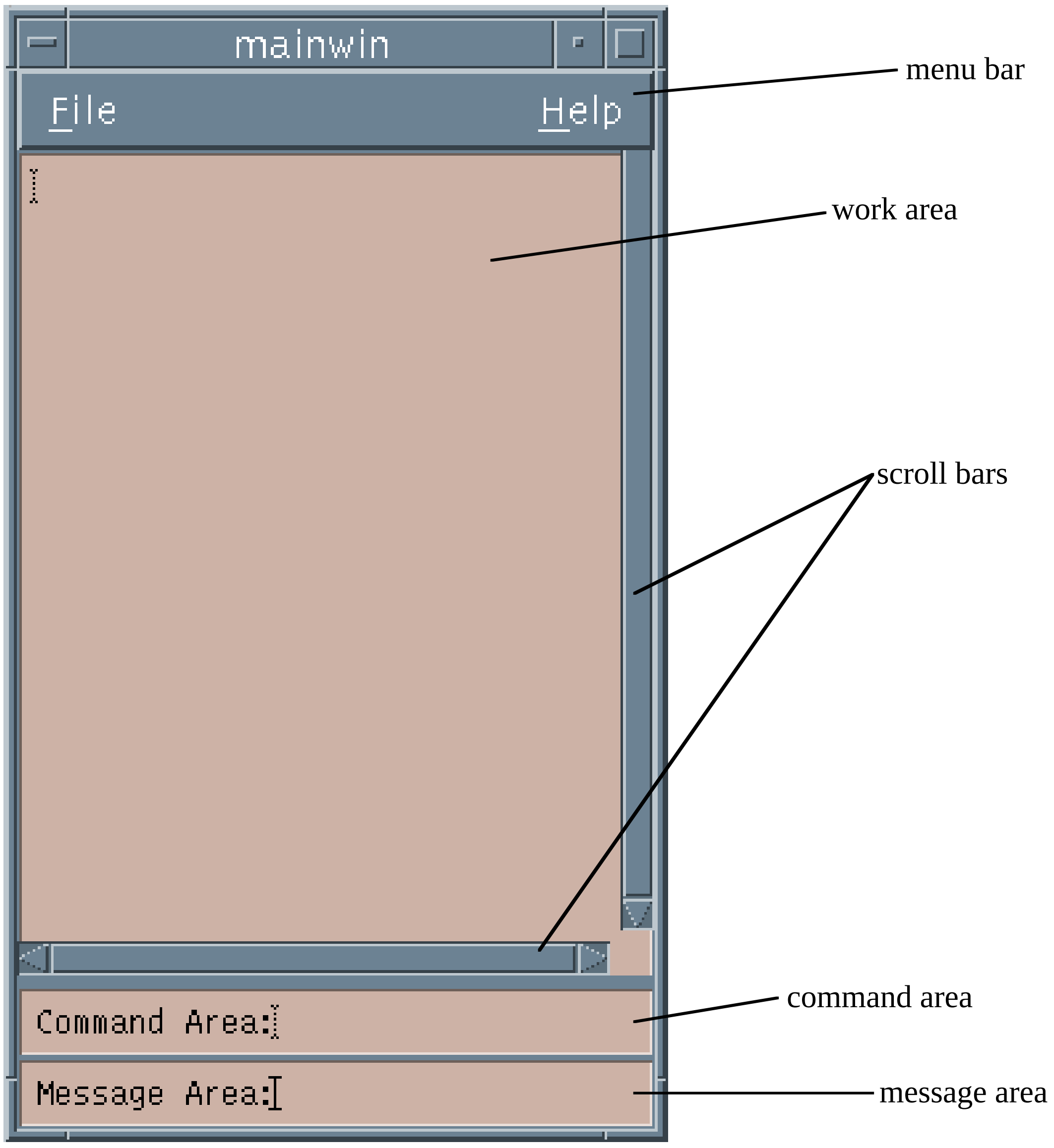


Figure 42.1: The MainWindow Widget with its Specialised Child Widgets

- The *message area* — where the application can report back to the user.

The MainWindow basically provides an efficient method of managing widgets as recommended by the *Motif Style Guide*(Chapter 52). In particular, the provision of a menu bar and work area is a convenient mechanism with which to drive many applications. It should be noted that the work area can be any widget (or composite hierarchy of widgets). The scrollbars, command and message area are optional.

In this Chapter, we will mainly concentrate our study on the relationship between the MainWindow and certain types of menus. We will also see how to put widgets in the work area MainWindow. We will see further applications of the MainWindow widget in the remainder of the book. The MainWindow will be used as a container widget in many example programs throughout the book.

42.1 The MainWindow widget

As we have previously stated, the MainWindow is typically used as the top level container for an application's child widgets. The simplest functional MainWindow may consist of a menu bar along the top of the application and some work area below, this is illustrated in Fig. 42.2. We omit the scroll bars and command and message areas for the time being.

Menu bar items are placed within the menu bar. You can use menu bar items to perform selections directly. However, more usually a *PullDown* menu is attached to a menu bar item allowing greater selection opportunities.

The function of the work area is to perform the main application's functions. The work area can be any widget class. We will look at aspects of this in later sections when we have studied more widget classes.

Initially, we will concentrate on how to create menus in a MainWindow.

42.2 The MenuBar

The creation of a fully functional pulldown MenuBar typical of most MainWindow based applications is fairly complex. We will, therefore, break it down into two stages.

1. We will we create a simple MenuBar that lets us select actions directly from the bar (*MenuBar items*).

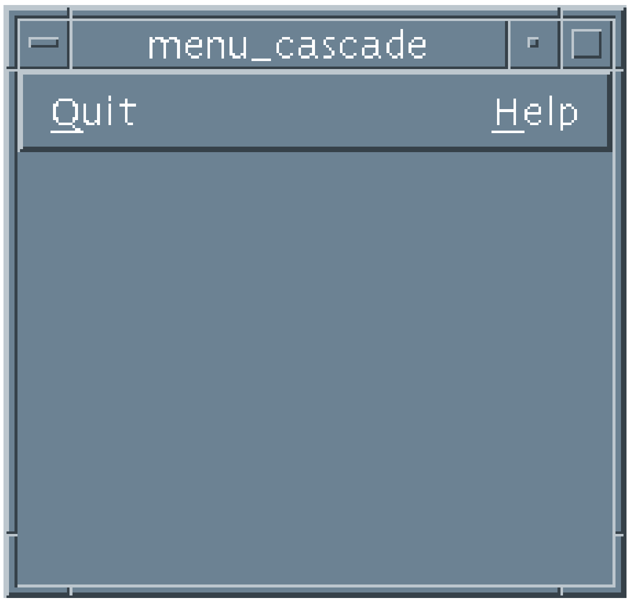


Figure 42.2: menu_cascade.c output

2. Having created simple MenuBar items we will attach pulldown menus to them.

42.2.1 A simple MenuBar

A MenuBar widget is really a RowColumn widget under another name. The way we create a MenuBar is to use one of the (several) `XmCreate`... or `XtVaCreate`... Menu options. For most of our applications, we will use the `XmCreateMenuBar()` function.

Having created a MenuBar we create MenuBar items by attaching widgets to the MenuBar in exactly the same fashion as described for the RowColumn widget (Chapter 41). The widgets we usually attach are CascadeButton widgets since we can hang pull down menus off these widgets.

Fig. 42.2 shows the output of the `menu_cascade.c` program that simply creates a simple MenuBar widget and attaches two CascadeButton widgets – `help` and `quit`. Minimal callback functions are associated with each CascadeButton.

The full program listing of `menu_cascade.c` is:

```
#include <Xm/Xm.h>
```

```
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>

/* Prototype callbacks */
void quit_call(void), help_call(void);

main(int argc, char **argv)

{
    Widget top_wid, main_w, menu_bar, quit, help;
    XtAppContext app;
    Arg arg[1];

    /* create application, main and menubar widgets */

    top_wid = XtVaAppInitialize(&app, "menu_cascade",
        NULL, 0, &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass,   top_wid,
        NULL);

    menu_bar = XmCreateMenuBar(main_w, "main_list",
        NULL, 0);
    XtManageChild(menu_bar);


     /* create quit widget + callback */

    quit = XtVaCreateManagedWidget( "Quit",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'Q',
        NULL);


    XtAddCallback(quit, XmNactivateCallback, quit_call, NULL);

    /* create help widget + callback */
    help = XtVaCreateManagedWidget( "Help",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'H',
        NULL);


    XtAddCallback(help, XmNactivateCallback, help_call, NULL);

    /* Tell the menubar which button is the help menu  */

    XtSetArg(arg[0],XmNmenuHelpWidget,help);
    XtSetValues(menu_bar,arg,1);


    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}


void quit_call()

{   printf("Quitting program\n");
    exit(0);
}

void help_call()

{   printf("Sorry, I'm Not Much Help\n");
}
```

The first steps of the `main` function should now be familiar — we initialise the application and create the `top_wid` top level widget.

A MainWindow widget, `main_win`, is then created as is a MenuBar, `menu_bar`. The `menu_bar` widget is a child of `main_win`. Finally, we attach two CascadeButton widgets, `quit` and `help`, as children of `menu_bar`. Note there is no work area created in this program.

A CascadeButton widget is usually used to attach a pulldown menu to the

MenuBar. This CascadeButton widget is similar to the PushButton widget and can have callback functions attached to its `activateCallback` function — illustrated in the `menu_cascade.c` program. However, it should be noted, that the main use of CascadeButton is to link a menu bar with a menu.

The program above simply attaches callback functions to the Cascade-Buttons. The callback functions do not do much except quit the program and print to standard output.

You can also associate a *mnemonic* to a particular menu selection. This means that you can use "hot keys" on the keyboard as a short cut to selection. You need to press `Meta` key plus the key in question.

In this program, we allow `Meta-Q` and `Meta-H` for the selection of Quit and Help. Note that Motif displays the appropriate Meta key by underlining the letter concerned on the menu bar (or menu item). The `XmNmnemonic` resource is used to select the appropriate keyboard short cut.

Apart from prescribing the use of the `Meta` key for the selection of menu items, the *Motif Style Guide* also insists that the Help MenuBar widget should always be placed on the right most side of the MenuBar (Fig. 42.2). This makes for easy location and selection of the help facility, should it exist.

The MenuBar resource, `XmNmenuHelpWidget`, is used to store the ID of the the appropriate widget (`help` in the above program).

42.2.2 PullDown Menus

Let us now develop things a little further by adding pulldown menus to our MenuBar. A pulldown menu looks like this:

In order to see how we create and use pulldown menu items we will develop a program, `menu_pull.c` that will create two pulldown menus:

- `Quit` which will contain a single item that lets us terminate our program.
- `Menu` which has 5 options that change the `XmNlabelString` of a label widget attached to the MainWindow as the work area.

We also attach the `Help` Cascade button as in the previous `menu_cascade.c` program.

We should now be familiar with the first few steps of this program: We create the top level widget hierarchy as usual, with the MainWindow wid-

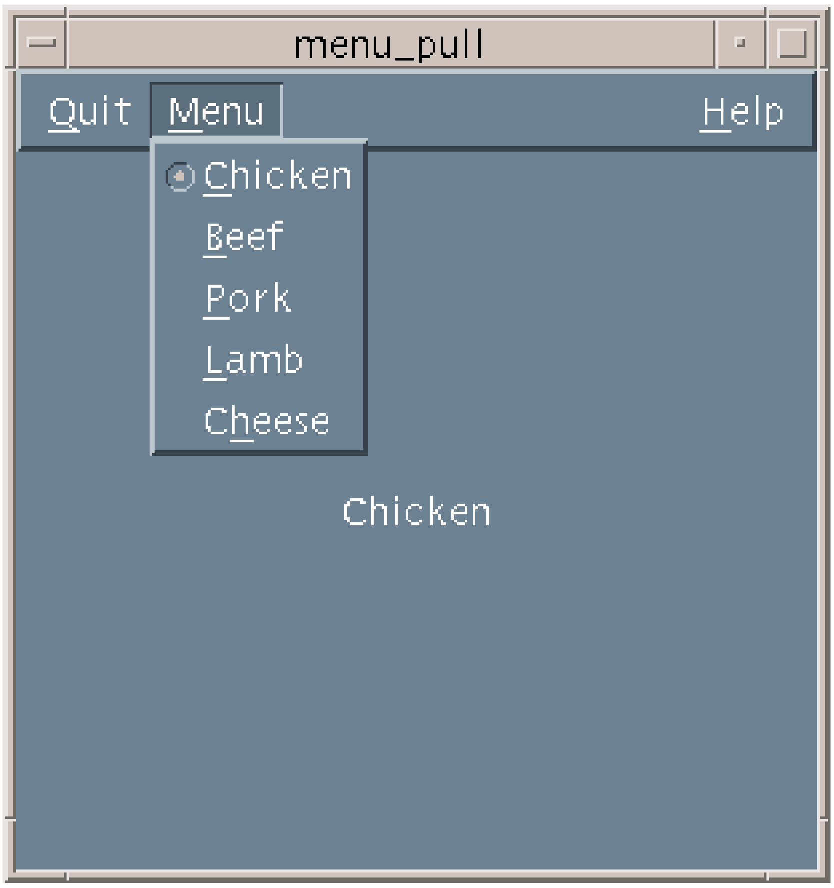


Figure 42.3: menu_pull.c output

get being a child of the application and a MenuBar widget a child of the MainWindow.

The complete listing of `menu_pull.c` is as follows:

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/Label.h>

/* prototype functions */

void quit_call(Widget , int),
       menu_call(Widget , int),
       help_call(void);


Widget label;
String food[] = { "Chicken",  "Beef", "Pork", "Lamb", "Cheese"};



main(int argc, char **argv)

{   Widget top_wid, main_w, help;
    Widget  menubar, menu, widget;
    XtAppContext app;
    XColor back, fore, spare;
    XmString  quit, menu_str,  help_str, chicken, beef, pork,
              lamb, cheese, label_str;

    int n = 0;
    Arg args[2];

    /* Initialize toolkit */
    top_wid = XtVaAppInitialize(&app, "Demos",
        NULL, 0, &argc, argv, NULL, NULL);
```

```
/* main window will contain a MenuBar and a Label  */
main_w = XtVaCreateManagedWidget("main_window",
    xmMainWindowWidgetClass,   top_wid,
    XmNwidth, 300,
    XmNheight, 300,
    NULL);

/* Create a simple MenuBar that contains three menus */
quit = XmStringCreateLocalized("Quit");
menu_str = XmStringCreateLocalized("Menu");
help_str = XmStringCreateLocalized("Help");


menubar = XmVaCreateSimpleMenuBar(main_w, "menubar",
    XmVaCASCADEBUTTON, quit, 'Q',
    XmVaCASCADEBUTTON, menu_str, 'M',
    XmVaCASCADEBUTTON, help_str, 'H',
    NULL);


XmStringFree(menu_str); /* finished with this so free */
XmStringFree(help_str);

/* First menu is the quit menu -- callback is quit_call() */

XmVaCreateSimplePulldownMenu(menubar, "quit_menu", 0, quit_call,
    XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
    NULL);
XmStringFree(quit);

/* Second menu is the food menu -- callback is menu_call() */
chicken = XmStringCreateLocalized(food[0]);
beef = XmStringCreateLocalized(food[1]);
pork = XmStringCreateLocalized(food[2]);
lamb = XmStringCreateLocalized(food[3]);
cheese = XmStringCreateLocalized(food[4]);
```

```
menu = XmVaCreateSimplePulldownMenu(menubar, "edit_menu", 1,
    menu_call,
    XmVaRADIOBUTTON, chicken, 'C', NULL, NULL,
    XmVaRADIOBUTTON, beef, 'B', NULL, NULL,
    XmVaRADIOBUTTON, pork, 'P', NULL, NULL,
    XmVaRADIOBUTTON, lamb, 'L', NULL, NULL,
    XmVaRADIOBUTTON, cheese, 'h', NULL, NULL,
    /* RowColumn resources to enforce */
    XmNradioBehavior, True,
    /* select radio behavior in Menu */
    XmNradioAlwaysOne, True,
    NULL);
XmStringFree(chicken);
XmStringFree(beef);
XmStringFree(pork);
XmStringFree(lamb);
XmStringFree(cheese);


/* Initialize menu so that "chicken" is selected. */
if (widget = XtNameToWidget(menu, "button_1"))
   { XtSetArg(args[n],XmNset, True);
     n++;
     XtSetValues(widget, args, n);
   }

n=0; /* reset n */

 /* get help widget ID to add callback */

help = XtVaCreateManagedWidget( "Help",
    xmCascadeButtonWidgetClass, menubar,
    XmNmnemonic, 'H',
    NULL);

XtAddCallback(help, XmNactivateCallback, help_call, NULL);
```

```
     /* Tell the menubar which button is the help menu  */

     XtSetArg(args[n],XmNmenuHelpWidget,help);
     n++;
     XtSetValues(menubar,args,n);
     n=0; /* reset n */





    XtManageChild(menubar);

   /* create a label text widget that will be "work area"
      selections from "Menu"  menu change label
      default label is item 0 */

    label_str = XmStringCreateLocalized(food[0]);

    label = XtVaCreateManagedWidget("main_window",
        xmLabelWidgetClass,   main_w,
        XmNlabelString, label_str,
        NULL);

     XmStringFree(label_str);

    /* set the label as the "work area" of the main window */
    XtVaSetValues(main_w,
        XmNmenuBar,    menubar,
        XmNworkWindow, label,
        NULL);


    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

/* Any item the user selects from the File menu calls this function.
   It will "Quit" (item_no == 0).  */
```

```
void
quit_call(Widget w, int item_no)

   /* w = menu item that was selected
      item_no = the index into the menu */

{


    if (item_no == 0) /* the "quit" item */
        exit(0);

   }


/* Called from any of the food "Menu" items.
   Change the XmNlabelString of the label widget.
   Note: we have to use dynamic setting with setargs().
 */

void  menu_call(Widget w, int item_no)


{
    int n =0;
    Arg args[1];

    XmString   label_str;

    label_str = XmStringCreateLocalized(food[item_no]);

    XtSetArg(args[n],XmNlabelString, label_str);
    ++n;
    XtSetValues(label, args, n);

}
```

```
void help_call()

{   printf("Sorry, I'm Not Much Help\n");
}
```

In this program we create the MenuBar widget with the convenience function `XmVaCreateSimpleMenuBar()`:

- The first two arguments of this function specify the *parent widget* (The MainWindow, `main_w`, in this case) and the *name* of the widget, respectively.
- The following arguments are a NULL terminated variable length list of resource name/value pairs.

  In `menu_pull.c`, we set up two CascadeButtons[1] by setting the `XmVACASCADEBUTTON` resource. Two arguments are associated with this resource:

  - A *label* specified by an `XmString`.
  - The *mnemonic* or *Meta* key associated with the CascadeButton.

  Note: we convert a `String` to an XmString with the `XmStringCreateLocalized()` function.

  It is good practice to free an XmString as soon as you have finished using it with the `XmStringFree()` command. Several "open" XmStrings can occupy a significant amount of application memory.

The creation of a pulldown menu is now relatively straightforward:

- We create a PullDownMenu widget

[1]XmVACASCADEBUTTON actually specifies CascadeButtonGadgets as children of the MenuBar, but you need not worry about the implications of this for now (*see* Section 39.3).

- Attach the PullDownMenu widget to a MenuBar item by simply making the PullDownMenu Widget a child of the MenuBar's CascadeButton.

Let us now look at how we create the `Quit` menu:

```
quit_w = XmVaCreateSimplePulldownMenu(menubar, "quit_menu",
            0, quit_call,
            XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
            NULL);
```

We have used the Motif convenience function `XmVaCreateSimplePulldownMenu()` to return a PulldownMenu widget. This function has several arguments:

- The first argument is the parent widget (`MenuBar` in this example).
- The second is the name given to the widget for resource lookup.
- The third argument is the integer *ID* that specifies which CascadeButton (from the MenuBar) the PulldownMenu is attached to.

  Thus, in this program an integer *ID* of `0` is attached to the `Quit` button and an *ID* of `1` would be attached to the `Menu` button.
- The fourth argument specifies the callback function associated with a menu choice.

  Note: We do not specify a callback with an `XtAddCallback()` call in this instance.
- A `NULL` terminated resource name/value list. The list specifies the type and values of PulldownMenu items. Many possible classes area allowed, including further CascadeButtons, RadioButtons, CheckButtons. This program creates PushButton Menu items:

  To specify a PushButton menu item, set a `XmVaPUSHBUTTON` resource list item and corresponding `XmString` label and Meta key accordingly for the list entry. `XmVaPUSHBUTTON` actually takes four arguments, the last two are only needed for advanced Motif use, so are not considered here — they are just set to `NULL`.

  The `Menu` menu is created in a similar way except that we have 5 menu items.

> We may have one, minor, problem when assigning Meta keys. This is illustrated for the `Menu` items since we cannot have the same Meta key for two menu selections. So `Meta-B` is chosen for Beef and `Meta-L` for Lamb, *etc.* However, Chicken and Cheese must be assigned different Meta keys, so we allocate `Meta-C` for Chicken and `Meta-h` for cheese selections.

The last thing we need to look at is how we find out which selection has been made in our program.

Each PulldownMenu has an associated callback function. The callback function of a pulldown has two parameters, which we must define.

- The widget that called the function.
- An index to the menu item selected (starts at 0).

So, in the Quit callback, `quit_call()`, we only have one possible selection (`item_no` must equal zero).

In the Menu callback, `menu_call()`, the index corresponds to a food item setting of Chicken, Beef, ... *etc.*.

Chapter 52 discusses aspects of the *Motif Style* guidelines for menus which also incorporate some general menu design issues.

42.2.3 Tear-off menus

Tear-off menus allow the user to remove (or *tear*) a menu off the MenuBar and keep it displayed in a small dialog window on the screen until the user closes it from the window menu. The *Motif Style Guide* (Chapter 52) prescribes this for menus that are frequently used in order to ease menu selection In order to make a menu a tear-off variety the `XmNtearOffModel` resource for a PullDownMenu widget needs to set to `XmTEAR_OFF_ENABLED`. If a menu is `XmTEAR_OFF_ENABLED` then its appearance is modified to include a small perforated line at the top of the menu.

42.3 Other MainWindow children

So far in this Chapter we have concentrated on the MenuBar and work area parts of the MainWindow. In this section we will develop a simple program, `test_for_echo.c`, which creates and uses the command and message areas of the MainWindow. We create a minimal MenuBar that simply allows you to quit the program. The work area created is a Label widget that does not perform any useful task in this example. The command and message areas created are both *TextField* widgets (*see* Chapter 44 for a complete description of the TextField widget). The command area receives (`String`) input and echoes it to the message area (Fig 42.4). The message area is not usually allowed to receive any user input. In this example we set the `XmNeditable` resource for the message area to be `False`

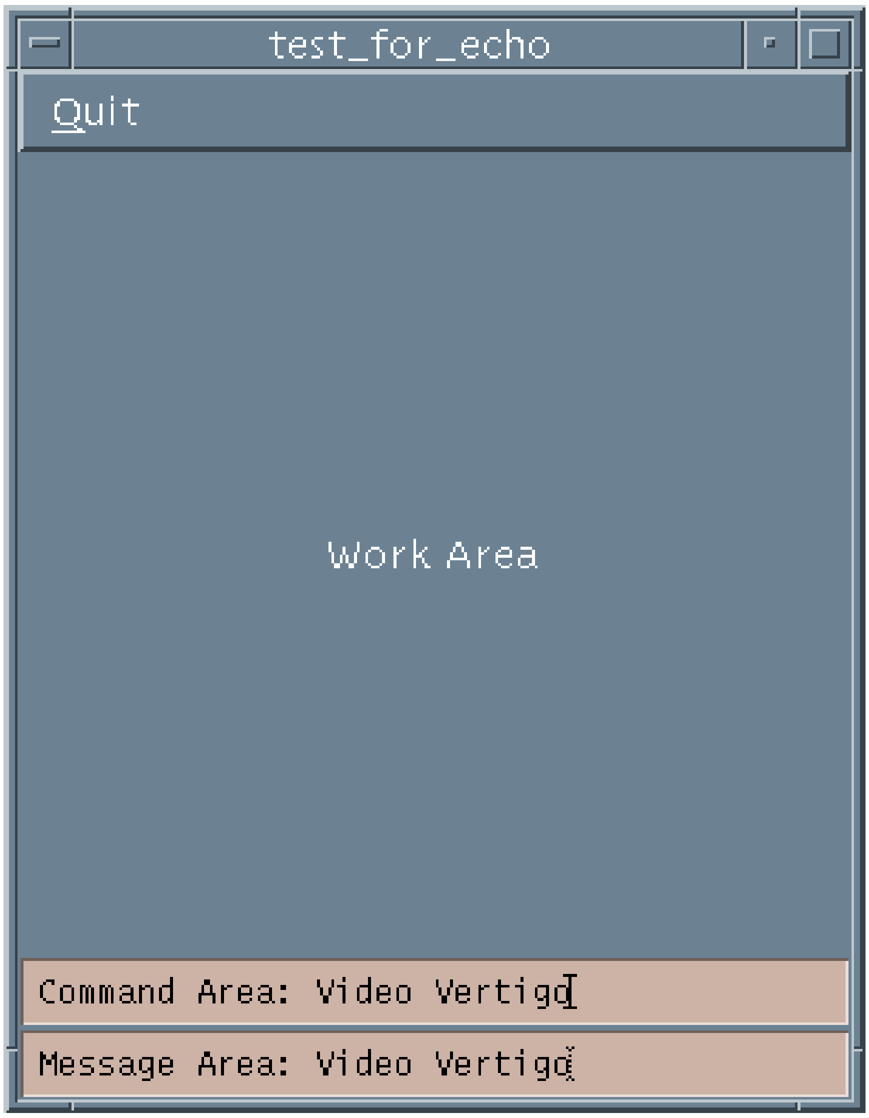
TextField, Label or Command widgets are typically employed as the command and message areas.

The test_for_echo.c program achieves this by:

- Creating a Menubar, Label and 2 TextField widgets and assigning their IDs to the MainWindow resources `XmNmenuBar`, `XmNworkWindow`, `XmNcommandWindow` and `XmNmessageWindow` respectively to form the complete Mainwindow widget (Fig 42.4).
- A callback, `cmd_cbk`, is attached to the `XmNactivateCallback` event for the command area widget.
- The `cmd_cbk` callback parses the input command area widget, `cmd_widget`, for the input string and replaces the message area widget, `msg_wid`, with this string.
- Section 42.3 in the following Chapter explains the text handling part of this program further.

The complete program listing for `test_for_echo.c` is as follows:

```
#include <Xm/MainW.h>
#include <Xm/Label.h>
#include <Xm/Command.h>
#include <Xm/TextF.h>

#include <stdio.h>
#include <string.h> /* For String Handling */

#define MAX_STR_LEN 30 /* Max Char length of Text Field */
```



```
Command Area: Video Vertigo
Message Area: Video Vertigo
```

Figure 42.4: The `test_for_echo.c` program output

```
/* Callback function prototypes */

void   cmd_cbk(), quit_cbk();

Widget   msg_wid;

char *cmd_label  = "Command Area: ";
char *msg_label  =  "Message Area: ";

int cmd_label_length;
int msg_label_length;


main(int argc,  char **argv)

{
    Widget        top_wid, main_win, menubar, menu,
                  label, cmd_wid;
    XtAppContext  app;
    XmString      label_str, quit;

    cmd_label_length = strlen(cmd_label);
    msg_label_length = strlen(msg_label);

    /* initialize toolkit and create top_widlevel shell */
    top_wid = XtVaAppInitialize(&app, "Main Window",
        NULL, 0, &argc, argv, NULL, NULL);


    /* Create MainWindow */

   main_win = XtVaCreateWidget("main_w",
        xmMainWindowWidgetClass, top_wid,
        XmNcommandWindowLocation, XmCOMMAND_BELOW_WORKSPACE,
        NULL);

    /* Create a simple MenuBar that contains one menu */
```

```
quit = XmStringCreateLocalized("Quit");
menubar = XmVaCreateSimpleMenuBar(main_win, "menubar",
    XmVaCASCADEBUTTON, quit, 'Q',
    NULL);


menu = XmVaCreateSimplePulldownMenu(menubar, "file_menu", 0,
    quit_cbk,
    XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
    NULL);

XmStringFree(quit);

/*  Manage Menubar */

XtManageChild(menubar);

/* create a label text widget that wil be work area  */
label_str = XmStringCreateLocalized("Work Area");

label = XtVaCreateManagedWidget("main_window",
    xmLabelWidgetClass,   main_win,
    XmNlabelString, label_str,
    XmNwidth, 1000,
    XmNheight, 800,
    NULL);

 XmStringFree(label_str);


/* Create the command area  */


cmd_wid = XtVaCreateWidget(  "Command",
    xmTextFieldWidgetClass,  main_win,
    XmNmaxLength, MAX_STR_LEN,
   NULL);
```

```
    XmTextSetString(cmd_wid,cmd_label);
    XmTextSetInsertionPosition(cmd_wid, cmd_label_length);

    XtAddCallback(cmd_wid, XmNactivateCallback, cmd_cbk);

    XtManageChild(cmd_wid);


 /* Create the message area  */

    msg_wid= XtVaCreateWidget(  "Message:",
        xmTextFieldWidgetClass,   main_win,
        XmNeditable, False,
        XmNmaxLength, MAX_STR_LEN,
        NULL);

    XmTextSetString(msg_wid,msg_label);

    XtManageChild(msg_wid);

/* set the label as the work, command and message areas
   of the main window */

    XtVaSetValues(main_win,
        XmNmenuBar,    menubar,
        XmNworkWindow, label,
        XmNcommandWindow, cmd_wid,
        XmNmessageWindow,msg_wid,
        NULL);

    XtManageChild(main_win);
    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

/* execute the command and redirect message area */
```

```
void cmd_cbk(Widget cmd_widget, XtPointer *client_data,
            XmAnyCallbackStruct  *cbs)
{
    char cmd[MAX_STR_LEN],msg[MAX_STR_LEN];

    XmTextGetSubstring(cmd_widget,cmd_label_length,
            MAX_STR_LEN - cmd_label_length, MAX_STR_LEN ,cmd);

    /* Append input message to Message area */

    XmTextReplace(msg_wid,msg_label_length, MAX_STR_LEN,cmd);

    /* Reset Command Area label  and insertion point*/

    XmTextSetString(cmd_widget, cmd_label);
    XmTextSetInsertionPosition(cmd_widget, cmd_label_length);
}

void  quit_cbk(Widget w, int item_no)

{    if (item_no == 0) /* the "quit" item */
        exit(0);
}
```

Chapter 43

Dialog Widgets

Any application needs to interact with the user. At the simplest level an application may need to inform, alert or warn the user about its current state. More advanced interaction may require the user to select or input data. Selecting files from a directory/file selection window is typical of an advanced example. Clearly, the provision of such interaction is the concern of the GUI. Motif provides a variety of *Dialog* widgets or *Dialogs* that facilitate most common user interaction requirements.

43.1 What are Dialogs?

Motif Dialog widgets usually comprise of the following components:

- A *dialog box* — Dialogs put up a box with a message in and may also prompt the user for some input.
- Three buttons may also be provided:
  - **Ok** and **Cancel**, used, perhaps, to acknowledge the message and remove the Dialog box. The Dialog usually remains on the screen until either the **Ok** or **Cancel** button has been pressed. The **Return** key can also be used to acknowledge the Dialog.
  - **Help**, some additional information may be provided by pressing this button.
- You have probably seen this in action with a text editor putting up a message of the form "Cannot open file ...."

More advanced Dialogs (*e.g. selection dialogs*) may significantly enhance this model.

Dialogs have many distinct uses, indeed the *Motif Style Guide* (Chapter 52) is specific in the use of each Dialog. The following Dialogs are provided by Motif:

WarningDialog — Warns User about a program mishap.

ErrorDialog — Alerts User to errors in the program.

InformationDialog — Program Information supplied.

PromptDialog — Allows user to supply data to program.

QuestionDialog — Yes/No type queries.

WorkingDialog — Notifies user if program is busy.

SelectionDialog — Selection from a list of options.

FileSelectionDialog — Specialised Selection Dialog to select directories and files.

BulletinBoardDialog — Allows customised Dialogs to be created.

43.2 Basic Dialog Management

To create a Dialog use one of the `XmCreate.....Dialog()` functions.

Dialogs do not usually appear immediately on screen after creation or when the the initial application GUI is realized. Indeed, if an application runs successfully certain Dialog widgets (Error or Warning Dialogs, in particular) may never be required. However, prudent applications should consider all practical avenues that an application would be expected to take and provide suitable information (via Dialogs) to the user.

It is advisable to create all Dialogs when the application is initialised and the overall GUI is setup. However, Dialogs will not be managed initially. Recall Section 38.7) when a widget is *unmanaged* by its parent it will always be invisible.

Therefore, Dialogs are usually created *unmanaged* and displayed when required in the program as described below:

- To make a Dialog appear in your program you must *manage* the widget, explicitly. `XtManageChild()` is the function typically employed.

- To make one disappear you must *unmanage* it, explicitly. `XtUnmanageChild()` is the function typically employed.

This method has the advantage that we only need to create a Dialog once and can then manage or unmanage it when necessary.

Most Motif Dialogs have default callback resources attached to the common **Ok** and **Cancel**. One consequence of these default callbacks is that they will *unmanage* the widgets from which they were called. However, if the application provides alternative callback (or other) functions then responsibility for correctly managing and unmanaging them is given over to the programmer.

The use of many dialogs is very similar. We will study a few specific Dialogs in detail.

43.3 The WarningDialog

This dialog is used to inform the user of a possible mistake in the program or in interaction with the program.

A typical example might be when you select the quit button (or menu item) to terminate the program — if this was selected mistakenly, and there is no warning prompt, you have problems.

The `dialog1.c` (Section 43.5) program attaches a pop-up WarningDialog when the quit menu option is selected (Fig 43.1). If you now select **OK** the program terminates, **Cancel** returns back to the program.

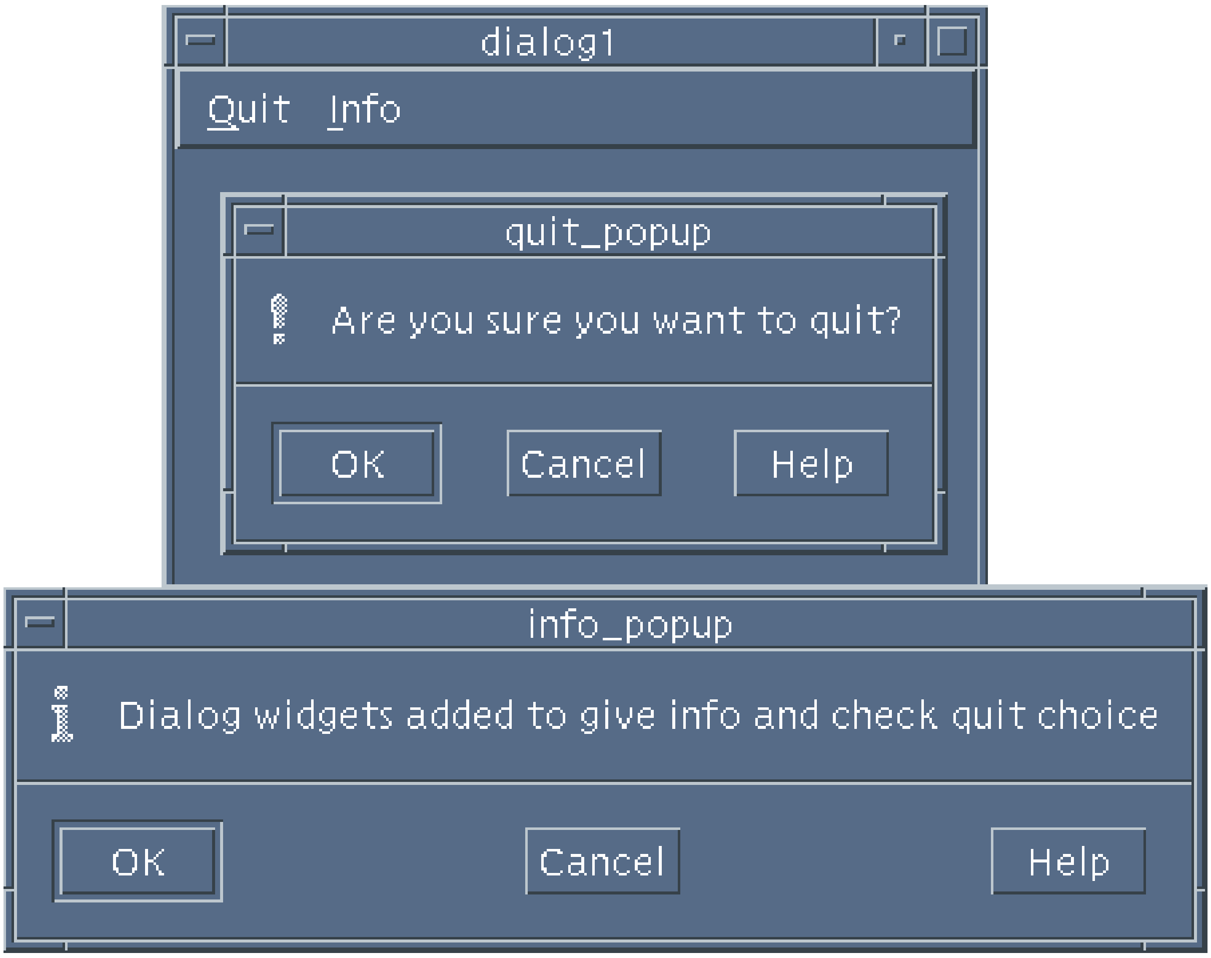
To Create a WarningDialog:

The function `create_dialogs()` in `dialog1.c` creates the WarningDialog in the following typical manner:

- All the warning Dialog contains is an `XmString`, which is the prompt to the user.

- For the Quit button we ask: "Are you sure you want to quit?".

- So, we set the `xm_string` variable accordingly.



Are you sure you want to quit?

Dialog widgets added to give info and check quit choice

Figure 43.1: WarningDialog and InformationDialog Widgets

- Set the `XmNmessageString` resource to the `xm_string` value.
- Create the WarningDialog widget with the `XmCreateWarningDialog()` function.
- Add callback functions to **Ok, Cancel** or **Help** buttons of Dialog.
- In this case, we only need to attach an application specific callback to the **Ok** button. The resource `XmNokCallback` therefore needs a function attached. Other button presses will use default callbacks. **NOTE:** The **Help** button does not do anything useful yet.

  The callback function `quit_pop_up()` simply needs to `XtManageChild()` the given Dialog widget so that it is displayed as required by the application.

43.4 The InformationDialog Widget

The InformationDialog Widget is essentially the same as the WarningDialog, except that InformationDialogs are intended to supply program information or help. In terms of Motif creation and management the widgets are identical except that the `XmCreateInformationDialog()` is used to create this class of Dialog. The main difference between the widgets to the user is visual. Motif employs different icons to distinguish between the two (Fig 43.1).

The program `dialog1.c` illustrates the creation and use of an InformationDialog Widget.

43.5 The `dialog1.c` program

This program uses Warning and Information Dialogs. These Dialogs are based on the MessageBox widget and so we must include `<Xm/MessageB.h>` header file.

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
```

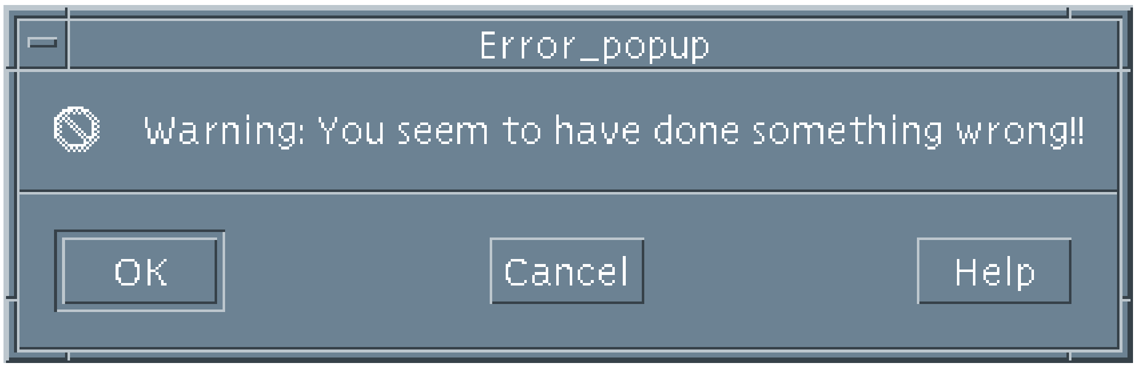
```
/* Prototype functions */

void create_dialogs(void);

/* callback for the pushbuttons.  pops up dialog */

void info_pop_up(Widget , char *, XmPushButtonCallbackStruct *),
       quit_pop_up(Widget , char *, XmPushButtonCallbackStruct *);

void info_activate(Widget ),
       quit_activate(Widget );

/* Global reference for dialog widgets */
Widget info, quit;
Widget info_dialog, quit_dialog;


main(int argc, char *argv[])
{
    XtAppContext app;
    Widget top_wid, main_w, menu_bar;

    top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
        &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass,   top_wid,
        XmNheight, 300,
        XmNwidth,300,
        NULL);


    menu_bar = (Widget) XmCreateMenuBar(main_w, "main_list", NULL, 0);
    XtManageChild(menu_bar);


    /* create quit widget + callback */
    quit = XtVaCreateManagedWidget( "Quit",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'Q',
        NULL);

    /* Callback has data passed to */

    XtAddCallback(quit, XmNactivateCallback, quit_pop_up, NULL);


    /* create help widget + callback */

    info = XtVaCreateManagedWidget( "Info",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'I',
        NULL);

    XtAddCallback(info, XmNactivateCallback, info_pop_up, NULL);


    /* Create but do not show (manage) dialogs */

    create_dialogs();


    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

void create_dialogs()

 {  /* Create but do not manage dialog widgets */

    XmString xm_string;
    Arg args[1];
    /* Create InformationDialog */

    /* Label the dialog */

    xm_string = XmStringCreateLocalized("Dialog widgets added to \
                give info and check quit choice");
    XtSetArg(args[0], XmNmessageString, xm_string);

    /* Create the InformationDialog */

    info_dialog = XmCreateInformationDialog(info, "info", args, 1);

    XmStringFree(xm_string);

    XtAddCallback(info_dialog, XmNokCallback, info_activate, NULL);

    /* Create Warning DIalog */

    /* label the dialog */

    xm_string =
      XmStringCreateLocalized("Are you sure you want to quit?");
    XtSetArg(args[0], XmNmessageString, xm_string);

    /* Create the WarningDialog */
    quit_dialog = XmCreateWarningDialog(quit, "quit", args, 1);

    XmStringFree(xm_string);

    XtAddCallback(quit_dialog, XmNokCallback, quit_activate, NULL);

}


void info_pop_up(Widget cascade_button, char *text,
                 XmPushButtonCallbackStruct *cbs)

{
    XtManageChild(info_dialog);
}

void quit_pop_up(Widget cascade_button, char *text,
                 XmPushButtonCallbackStruct *cbs)

{
    XtManageChild(quit_dialog);
}

/* callback routines for dialogs */

void info_activate(Widget dialog)
{
    printf("Info Ok was pressed.\n");
}

void quit_activate(Widget dialog)
{
    printf("Quit Ok was pressed.\n");
    exit(0);
}
```

43.6 Error, Working and Question Dialogs

These 3 Dialogs are similar to both the Information and Warning Dialogs. They are created and used in similar fashions. The main difference again being the icon used to depict the Dialog class as illustrated in Figs. 43.2 — 43.4.

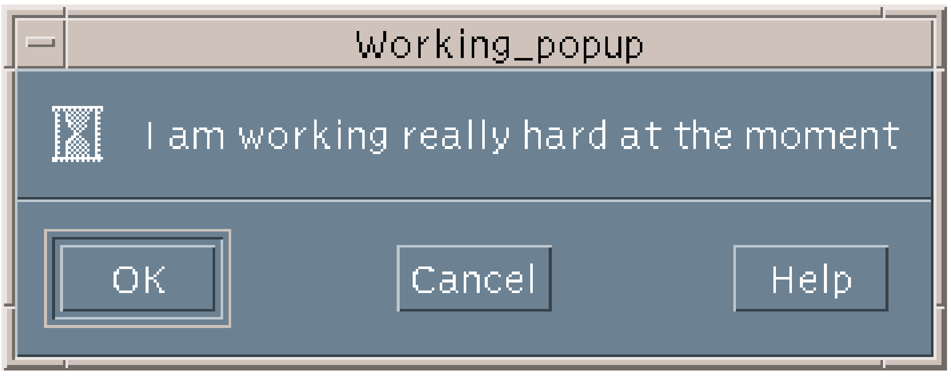
43.7 Unwanted Dialog Buttons

When you create a Dialog, Motif will create 3 buttons by default — **Ok, Cancel** and **Help**. There are many occasions when it is not natural to require the use of three buttons within an application. For instance in `dialog1.c` we only really need the user to acknowledge the InformationDialog and no



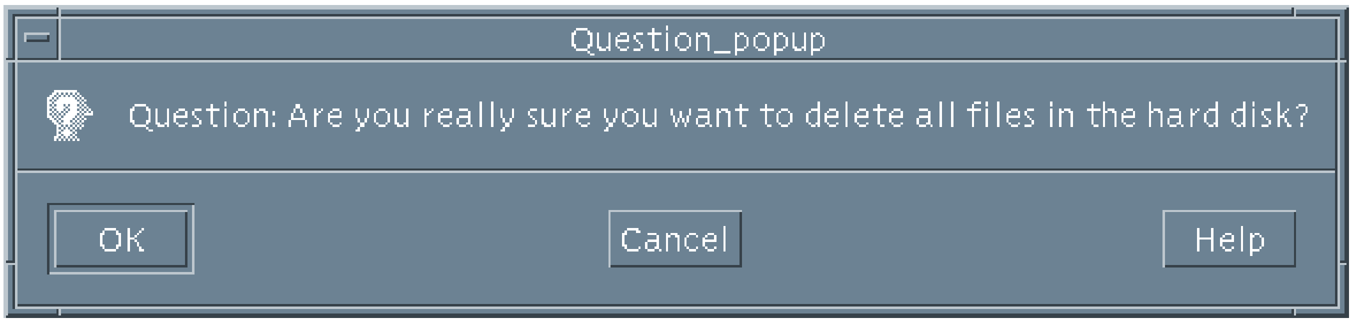
Warning: You seem to have done something wrong!!

Figure 43.2: ErrorDialog Widget



I am working really hard at the moment

Figure 43.3: WorkingDialog Widget



Question: Are you really sure you want to delete all files in the hard disk?

Figure 43.4: QuestionDialog Widget

user should need any help to choose whether to quit our program.

Motif provides a mechanism to disable unwanted buttons in a Dialog.

To remove a button:

- Use `XmMessageBoxGetChild()` (or similar function) to get the button widget's ID for the given dialog widget and button type.
- `XmDIALOG_OK_BUTTON`, `XmDIALOG_CANCEL_BUTTON` or `XmDIALOG_HELP_BUTTON` specify the type of the button.
- Unmanage the button widget.

An example where we disable the **Cancel** and **Help** buttons on the InformationDialog and the **Help** button on the WarningDialog from the Dialogs created in program `dialog1.c` is shown in Fig. 43.5. The code that performs the task of deleting the **Help** from a widget, `dialog` is as follows:

```
Widget remove;

remove = XmMessageBoxGetChild(dialog, XmDIALOG_HELP_BUTTON);

XtUnmanageChild(remove);
```

43.8 The PromptDialog Widget

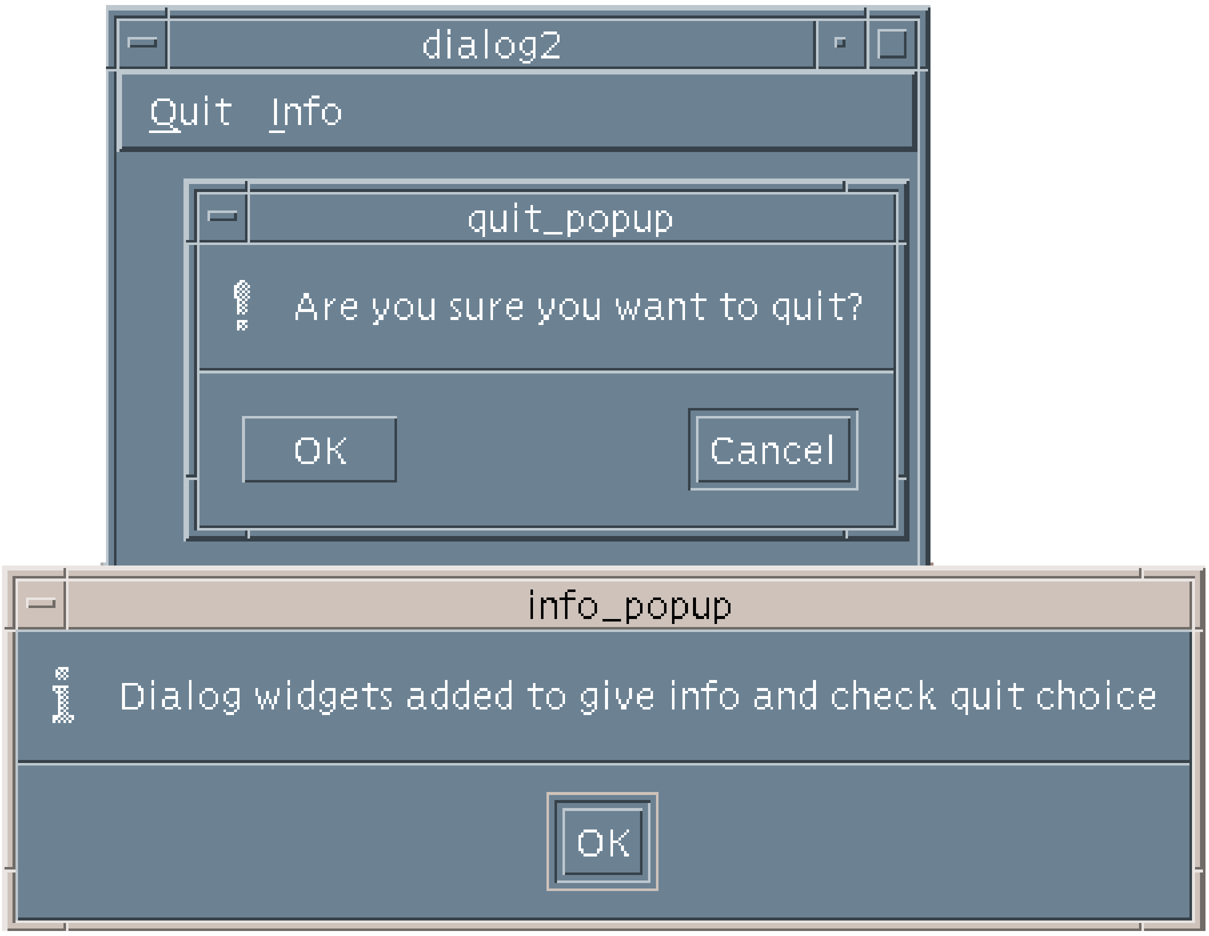
The PromptDialog widget is slightly more advanced than the classes of Dialog widgets encountered so far. This widget allows the user to enter text (Fig. 43.6).

The function `XmCreatePromptDialog()` instantiates the Dialog. Typically two resources of a PromptDialog widget are required to be set. These resources are

`XmNselectionLabelString` — the prompt message, an `XmString` data type.

`XmNtextString` — the default response `XmString` which may be empty.

Note: A Prompt Dialog is based on the SelectionBox widget and so we must include `<Xm/SelectioB.h>` header file.



Are you sure you want to quit?

Dialog widgets added to give info and check quit choice

Figure 43.5: Removed Dialog Button

43.8.1 The `prompt.c` program

This program, an extension of `dialog1.c`, creates a PromptDialog into which the user can enter text. The PromptDialog first displayed by this program is shown in Fig. 43.6. The text is echoed in an InformationDialog which is created in a Prompt callback function, `prompt_activate()`.

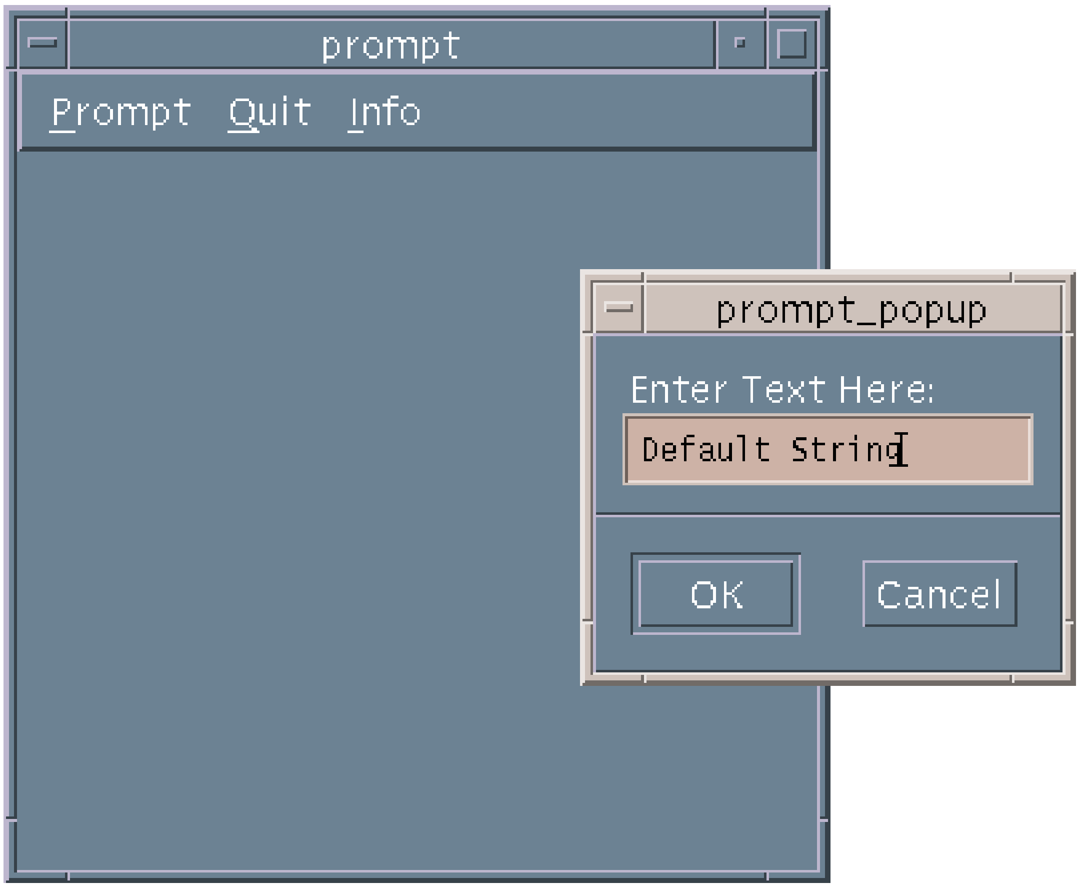


Figure 43.6: PromptDialog Widget

A PromptDialog Callback has the following structure

```
 void prompt_callback(Widget widget, XtPointer client_data,
XmSelectionBoxCallbackStruct *selection)
```

Normally, we will only be interested in obtaining the string entered to the

PromptDialog. An element of the XmSelectionBoxCallbackStruct, value holds the (XmString data type) value.

In prompt.c, the callback for the prompt dialog (activated with **Ok** button) — prompt_activate().

This function uses the selection->value XmString to set up the InformationDialog message String.

Note, that since a PromptDialog is a SelectionBox widget type we must use XmSelectionBoxGetChild() to find any buttons we may wish to remove from the PromptDialog. In prompt.c we remove the **Help** button in this way.

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <Xm/SelectioB.h>

/* Callback and other function prototypes */

void ScrubDial(Widget, int);
void info_pop_up(Widget  , char *, XmPushButtonCallbackStruct *),
     quit_pop_up(Widget  , char *, XmPushButtonCallbackStruct *),
     prompt_pop_up(Widget  , char *, XmPushButtonCallbackStruct *);
void prompt_activate(Widget , caddr_t,
                   XmSelectionBoxCallbackStruct *);
void quit_activate(Widget);

Widget top_wid;

main(int argc, char *argv[])

{
    XtAppContext app;
    Widget main_w, menu_bar, info, prompt, quit;


    top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
        &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass,   top_wid,
        XmNheight, 300,
        XmNwidth,300,
        NULL);


    menu_bar = XmCreateMenuBar(main_w, "main_list",
        NULL, 0);
    XtManageChild(menu_bar);

    /* create prompt widget + callback */
    prompt = XtVaCreateManagedWidget( "Prompt",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'P',
        NULL);

    /* Callback has data passed to */
    XtAddCallback(prompt, XmNactivateCallback,
                  prompt_pop_up, NULL);


    /* create quit widget + callback */

    quit = XtVaCreateManagedWidget( "Quit",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'Q',
        NULL);


    /* Callback has data passed to */

    XtAddCallback(quit, XmNactivateCallback, quit_pop_up,
                  "Are you sure you want to quit?");
    /* create help widget + callback */

    info = XtVaCreateManagedWidget( "Info",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'I',
        NULL);

    XtAddCallback(info, XmNactivateCallback, info_pop_up,
            "Select Prompt Option To Get Program Going.");

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app); }

 void prompt_pop_up(Widget cascade_button, char *text,
XmPushButtonCallbackStruct *cbs)

{   Widget dialog, remove;
    XmString xm_string1, xm_string2;
    Arg args[3];

    /* label the dialog */
    xm_string1 = XmStringCreateLocalized("Enter Text Here:");
    XtSetArg(args[0], XmNselectionLabelString, xm_string1);
    /* default text string  */
    xm_string2 = XmStringCreateLocalized("Default String");



    XtSetArg(args[1], XmNtextString, xm_string2);
   /* set up default button for cancel callback */
    XtSetArg(args[2], XmNdefaultButtonType,
                    XmDIALOG_CANCEL_BUTTON);

    /* Create the WarningDialog */
    dialog = XmCreatePromptDialog(cascade_button, "prompt",
                              args, 3);

    XmStringFree(xm_string1);
    XmStringFree(xm_string2);
    XtAddCallback(dialog, XmNokCallback, prompt_activate,
                  NULL);
   /* Scrub Prompt Help Button */
    remove = XmSelectionBoxGetChild(dialog,
                      XmDIALOG_HELP_BUTTON);

    XtUnmanageChild(remove);  /* scrub HELP button */

    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone); }

 void info_pop_up(Widget cascade_button, char *text,
XmPushButtonCallbackStruct *cbs)

{   Widget dialog;
    XmString xm_string;
    extern void info_activate();
    Arg args[2];

    /* label the dialog */
    xm_string = XmStringCreateLocalized(text);
    XtSetArg(args[0], XmNmessageString, xm_string);
    /* set up default button for OK callback */
    XtSetArg(args[1], XmNdefaultButtonType,
             XmDIALOG_OK_BUTTON);


    /* Create the InformationDialog as child of
       cascade_button passed in */
    dialog = XmCreateInformationDialog(cascade_button,
                                "info", args, 2);

    ScrubDial(dialog, XmDIALOG_CANCEL_BUTTON);
    ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

    XmStringFree(xm_string);

    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone);
}

 void quit_pop_up(Widget cascade_button, char *text,
XmPushButtonCallbackStruct *cbs)

{   Widget dialog;
    XmString xm_string;
    Arg args[1];

    /* label the dialog */
    xm_string = XmStringCreateLocalized(text);
    XtSetArg(args[0], XmNmessageString, xm_string);
    /* set up default button for cancel callback */
    XtSetArg(args[1], XmNdefaultButtonType,
                   XmDIALOG_CANCEL_BUTTON);

    /* Create the WarningDialog */
    dialog = XmCreateWarningDialog(cascade_button, "quit",
                                 args, 1);

    ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

    XmStringFree(xm_string);

    XtAddCallback(dialog, XmNokCallback, quit_activate,
                  NULL);

    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone);
}

 /* routine to remove a DialButton from a Dialog */

void ScrubDial(Widget wid, int dial)

{  Widget remove;
```

```
   remove = XmMessageBoxGetChild(wid, dial);

   XtUnmanageChild(remove);
}


 /* callback function for Prompt activate */

void prompt_activate(Widget widget, XtPointer client_data,
XmSelectionBoxCallbackStruct *selection)

{   Widget dialog;
    Arg args[2];
    XmString xm_string;

    /* compose InformationDialog output string */
    /* selection->value holds XmString entered to prompt */

    xm_string = XmStringCreateLocalized("You typed: ");
    xm_string = XmStringConcat(xm_string,selection->value);


    XtSetArg(args[0], XmNmessageString, xm_string);
    /* set up default button for OK callback */
    XtSetArg(args[1], XmNdefaultButtonType,
             XmDIALOG_OK_BUTTON);

    /* Create the InformationDialog to echo
       string grabbed from prompt */
    dialog = XmCreateInformationDialog(top_wid,
           "prompt_message", args, 2);

    ScrubDial(dialog, XmDIALOG_CANCEL_BUTTON);
    ScrubDial(dialog, XmDIALOG_HELP_BUTTON);

    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone);
}
```

```
/* callback routines for quit ok dialog */


void quit_activate(Widget dialog) {
    printf("Quit Ok was pressed.\n");
    exit(0);
}
```

43.9 Selection and FileSelection Dialogs

The purpose of both these widgets is to allow the user to select from a list (or in set of lists) displayed within the Dialog. The creation and use of Selection and FileSelection Dialogs is similar. The FileSelectionDialog (Fig. 43.7) allows the selection of files from a directory which has use in many applications (text editors, graphics programs *etc.*) The FileSelection Dialog provides a means for merely browsing directories and selecting file *names.* It is up to the application to read/write the file, or to use the file name appropriately. The SelectionDialog allows for more general selection. We will study the FileSelectionDialog as it is more complex and it is also more commonly used.

To create a FileSelectionDialog, the `XmCreateFileSelectionDialog()` function is commonly used. You must include the `<Xm/FileSB.h>` header file.

A FileSelectionDialog has many resources you can set to control the search of files: (All resources are XmString data types except where indicated.)

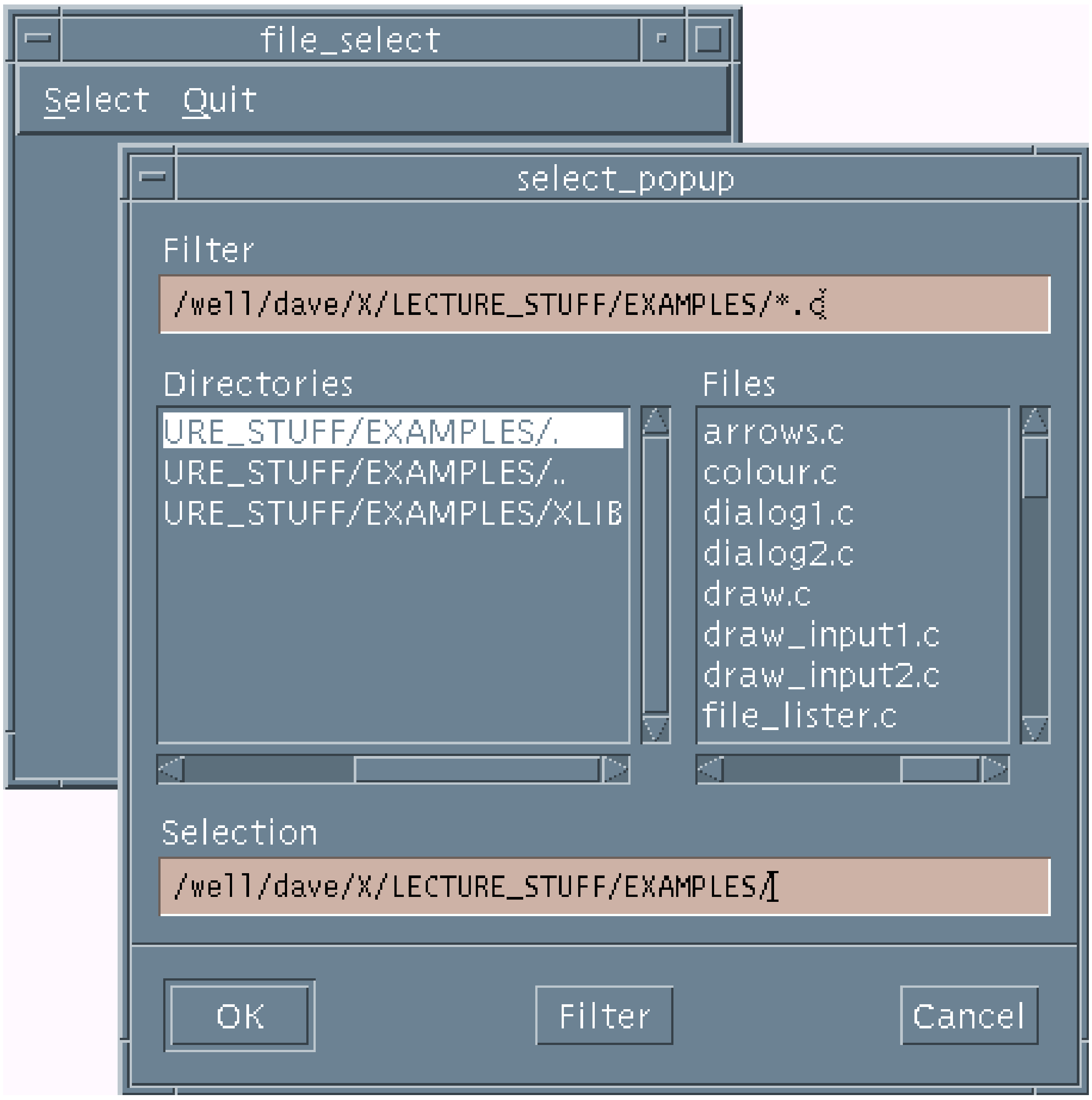
XmNdirectory — The directory from where files are (initially) listed. The *default* directory is the *current working directory.*

XmNdirListLabelString — The label displayed above the directory listing in Dialog.

XmNdirMask — The mask used to filter out certain files. *E.g.* in `file_select.c` this mask is set to `*.c` so as only to list C source files in the dialog.

XmNdirSpec — the name of the file (to be) chosen.

XmNfileListLabelString — The label displayed above the file list.



```
/well/dave/X/LECTURE_STUFF/EXAMPLES/*.c
```

Directories:

```
URE_STUFF/EXAMPLES/.
URE_STUFF/EXAMPLES/..
URE_STUFF/EXAMPLES/XLIB
```

Files:

```
arrows.c
colour.c
dialog1.c
dialog2.c
draw.c
draw_input1.c
draw_input2.c
file_lister.c
```

Selection:

```
/well/dave/X/LECTURE_STUFF/EXAMPLES/
```

Figure 43.7: The FileSelectionDialog Widget

XmNfileTypeMask — The type of file to be listed (*char* type). `XmFILE_REGULAR, XmFILE_DIRECTORY, XmFILE_TYPE_ANY` are conveniently predefined macros in `<Xm/FileSB.h>`.

XmNfilterLabelString — The label displayed above the filter list.

The search directory, directory mask and others can be altered from within the Dialog window.

The FileSelectionDialog has many child widgets under its control. It is sometimes useful to take control of these child widget in order to have greater control of their resources or callback or even to remove (`XtUnmanageChild()`) one. The function `XmFileSelectionBoxGetChild()` is used to return the ID of a specified child widget. The function takes two arguments:

Widget — the parent widget.

Child — an unsigned char. Possible values for this argument are defined in `<Xm/FileSB.h>` and include:

```
XmDIALOG_APPLY_BUTTON,   XmDIALOG_LIST,
XmDIALOG_CANCEL_BUTTON,  XmDIALOG_LIST_LABEL,
XmDIALOG_DEFAULT_BUTTON, XmDIALOG_OK_BUTTON,
XmDIALOG_DIR_LIST,       XmDIALOG_SELECTION_LABEL,
XmDIALOG_DIR_LIST_LABEL, XmDIALOG_SEPARATOR,
XmDIALOG_FILTER_LABEL,   XmDIALOG_TEXT,
XmDIALOG_FILTER_TEXT,    XmDIALOG_WORK_AREA,
XmDIALOG_HELP_BUTTON.
```

43.9.1 The `file_select.c` program

The program `file_select.c` simply looks for C source files in a directory — the `XmNdirMask` resource is set to filter out only `*.c` files. If a file is selected it's *listing* is printed to standard output.

```
#include <stdio.h>

#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
```

```
#include <Xm/MessageB.h>
#include <Xm/PushB.h>
#include <Xm/FileSB.h>


/* prototype callbacks and other functions */
void  quit_pop_up(Widget , char *,
                  XmPushButtonCallbackStruct *),
void  select_pop_up(Widget , char *,
                   XmPushButtonCallbackStruct *);
void ScrubDial(Widget, int);
void select_activate(Widget , XtPointer ,
                     XmFileSelectionBoxCallbackStruct *)
void quit_activate(Widget)
void cancel(Widget , XtPointer ,
            XmFileSelectionBoxCallbackStruct *);
void error(char *, char *);

File *fopen();

Widget top_wid;


main(int argc, char *argv[])

{
    XtAppContext app;
    Widget main_w, menu_bar, file_select, quit;


    top_wid = XtVaAppInitialize(&app, "Demos", NULL, 0,
        &argc, argv, NULL, NULL);

    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass,   top_wid,
        XmNheight, 300,
        XmNwidth,300,
        NULL);
    menu_bar = XmCreateMenuBar(main_w, "main_list",
        NULL, 0);
    XtManageChild(menu_bar);


    /* create prompt widget + callback */

    file_select = XtVaCreateManagedWidget( "Select",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'S',
        NULL);

    /* Callback has data passed to */
    XtAddCallback(file_select, XmNactivateCallback,
                  select_pop_up, NULL);

    /* create quit widget + callback */
    quit = XtVaCreateManagedWidget( "Quit",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'Q',
        NULL);

    /* Callback has data passed to */
    XtAddCallback(quit, XmNactivateCallback, quit_pop_up,
                  "Are you sure you want to quit?");

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

void select_pop_up(Widget cascade_button, char *text,
                   XmPushButtonCallbackStruct *cbs)

{   Widget dialog, remove;
    XmString mask;
    Arg args[1];

    /* Create the FileSelectionDialog */
    mask  = XmStringCreateLocalized("*.c");
    XtSetArg(args[0], XmNdirMask, mask);

    dialog = XmCreateFileSelectionDialog(cascade_button,
           "select", args, 1);
    XtAddCallback(dialog, XmNokCallback, select_activate,
                  NULL);
    XtAddCallback(dialog, XmNcancelCallback, cancel, NULL);

    remove = XmSelectionBoxGetChild(dialog,
           XmDIALOG_HELP_BUTTON);

    XtUnmanageChild(remove); /* delete HELP BUTTON */


    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone);
}

void quit_pop_up(Widget cascade_button, char *text,
                 XmPushButtonCallbackStruct *cbs)

{   Widget dialog;
    XmString xm_string;
    Arg args[2];

    /* label the dialog */
    xm_string = XmStringCreateLocalized(text);
    XtSetArg(args[0], XmNmessageString, xm_string);
    /* set up default button for cancel callback */
    XtSetArg(args[1], XmNdefaultButtonType,
             XmDIALOG_CANCEL_BUTTON);

    /* Create the WarningDialog */
    dialog = XmCreateWarningDialog(cascade_button, "quit",
           args, 2);

    ScrubDial(dialog, XmDIALOG_HELP_BUTTON);
    XmStringFree(xm_string);

    XtAddCallback(dialog, XmNokCallback, quit_activate,
                  NULL);

    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone);
}

 /* routine to remove a DialButton from a Dialog */

void ScrubDial(Widget wid, int dial)

{  Widget remove;

   remove = XmMessageBoxGetChild(wid, dial);
   XtUnmanageChild(remove);
}

/* callback function for Prompt activate */

void select_activate(Widget widget, XtPointer client_data,
                     XmFileSelectionBoxCallbackStruct *selection)

{   /* function opens file (text) and prints to stdout */

    FILE *fp;
    char *filename, line[200];


    XmStringGetLtoR(selection->value,
 XmSTRING_DEFAULT_CHARSET, &filename);


    if ( (fp = fopen(filename,"r")) == NULL)
      error("CANNOT OPEN FILE", filename);
  else
```

```
    {  while ( !feof(fp) )
          { fgets(line,200,fp);
            printf("%s\n",line);
          }
       fclose(fp);
    }
}



void cancel(Widget widget, XtPointer client_data,
            XmFileSelectionBoxCallbackStruct *selection)

{ XtUnmanageChild(widget);  /* undisplay widget */
}

void error(char *s1, char *s2)

{  /* prints error to stdout */


   printf("%s: %s\n", s1, s2);
   exit(-1);

}



 /* callback routines for quit ok dialog */


void quit_activate(Widget dialog)

{
    printf("Quit Ok was pressed.\n");
    exit(0);

}
```

43.10 User Defined Dialogs

Motif allows the programmer to create new customised Dialogs. BulletinBoardDialogs and FormDialogs let you place widgets within them in a similar fashion to their corresponding BulletinBoard and Form widgets. We will, therefore, not deal with these further in this text.

43.11 Exercises

Exercise 43.1 *Rewrite the* `error()` *function of the* `file_select.c` *program so that errors trapped by this program are displayed in an ErrorDialog widget and not simply printed to standard output.*

Chapter 44

Text Widgets

Text editing is a key task in many applications. For example, Single-line editors are a convenient and flexible means of string data entry for many applications. Indeed, the FileSelectionDialog widget (Chapter 43) and other composite widgets have a single text widget as constituent components. More complete multi-line text entry may also be required for many applications.

Motif, conveniently provides a fully functional text widget. This saves the application programmer a lot of work, since tasks such as *cut and paste* editing, text search and insertion are provided within the widget class.

Note: Coupled with other advanced facilities such as FileSelection widgets *etc.*, we could easily assemble our own fully working text editor application program from component widget classes and little other code.

Motif 1.2 provides two classes of text widgets:

Text widget — A fully functional multiline windows based text editor.

TextField — A single line text editor.

Both the above widgets use the (standard C) `String` data type as the base structure for all text operations. This is different from most other Motif widgets. Motif 2.0 provides an additional text widget, *CSText*, which is basically similar to the *Text* widget except that the `XmString` data type is used in text processing.

We will study the Text widget in detail in this Chapter. The TextField is, essentially, a simpler version of this and will therefore only be addressed when appropriate. In fact, we can actually make the **Text** widget a single line type by setting the resource `XmNeditMode` to `XmSINGLE_LINE_EDIT`.

44.1 Text Widget Creation

There are a variety of ways to create a Text widget:

- We can use the usual `XmCreateText()` or `XtVaCreateManagedWidget()` methods.
- Usually we will need to use a *scrolled* window since the text we are editing may exceed the window size (Fig 44.1).
  - To create a ScrolledText widget use `XmCreateScrolledText()`.
- We must include the `<Xm/Text.h>` header file for all Text widget applications. There are corresponding `<Xm/TextF.h>` and `<Xm/CSText.h>` header files for the TextField amd CSText widgets respectively.

There are various resources that can be usefully set for a Text widget:

XmNrows — The visible number of rows.

XmNcolumns — The visible number of columns.

XmNeditable — `True` or `False`: determines whether editing of displayed text is allowed.

XmNscrollHorizontal — `True` or `False`: Turn scrolling On/Off in this direction.

XmNscrollVertical — `True` or `False`: Turn scrolling On/Off in this direction.

44.2 Putting text into a Text Widget

The Text widget is a *dynamic* structure and text may be inserted into the widget at any time. There are many text editing and insertion functions that will be introduced shortly. The simplest operation is actually setting the text that will be used by the Text widget.

The function `XmTextSetString()` puts a specified *ordinary (C type)* string into a specified widget. It has two arguments:

Widget — the Text widget,

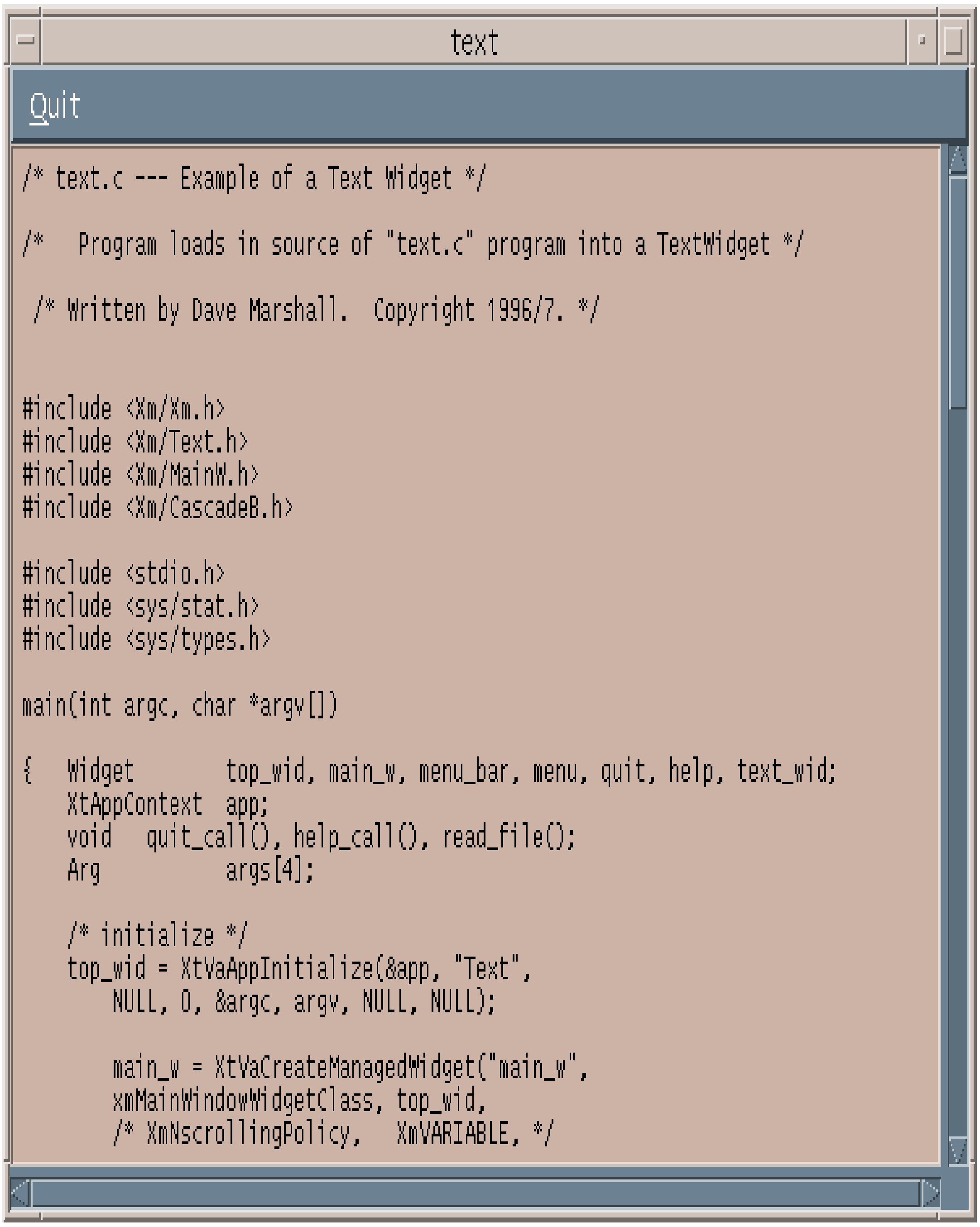
String — The text being placed in the widget.

44.3 Example Text Program - `text.c`

This program is a fairly simple example of the `Text` widget in use.

- It creates a `ScrolledText` widget and places the source code of the `text.c` program in the `Text` widget (Fig. 44.1).
- **No** editing is allowed.

```
#include <Xm/Xm.h>
#include <Xm/Text.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>


/* Prototype Callback and other functions */

void  quit_call(),
      help_call(),
      read_file(Widget);

main(int argc, char *argv[])

{   Widget        top_wid, main_w, menu_bar,
                  menu, quit, help, text_wid;
    XtAppContext  app;
    Arg           args[4];

    /* initialize */
    top_wid = XtVaAppInitialize(&app, "Text",
        NULL, 0, &argc, argv, NULL, NULL);

        main_w = XtVaCreateManagedWidget("main_w",
        xmMainWindowWidgetClass, top_wid,
```



```
/* text.c --- Example of a Text Widget */

/*   Program loads in source of "text.c" program into a TextWidget */

 /* Written by Dave Marshall.  Copyright 1996/7. */


#include <Xm/Xm.h>
#include <Xm/Text.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>

#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

main(int argc, char *argv[])

{   Widget       top_wid, main_w, menu_bar, menu, quit, help, text_wid;
    XtAppContext  app;
    void   quit_call(), help_call(), read_file();
    Arg          args[4];

    /* initialize */
    top_wid = XtVaAppInitialize(&app, "Text",
        NULL, 0, &argc, argv, NULL, NULL);

        main_w = XtVaCreateManagedWidget("main_w",
        xmMainWindowWidgetClass, top_wid,
        /* XmNscrollingPolicy,   XmVARIABLE, */
```

Figure 44.1: ScrolledText Widget

```
        /* XmNscrollingPolicy,   XmVARIABLE, */
        NULL);

    menu_bar = XmCreateMenuBar(main_w, "main_list",
        NULL, 0);
    XtManageChild(menu_bar);

     /* create quit widget + callback */

    quit = XtVaCreateManagedWidget( "Quit",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'Q',
        NULL);

    XtAddCallback(quit, XmNactivateCallback,
        quit_call, NULL);

    /* Create ScrolledText -- this is work area for the
       MainWindow */

    XtSetArg(args[0], XmNrows,      30);
    XtSetArg(args[1], XmNcolumns,   80);
    XtSetArg(args[2], XmNeditable,  False);
    XtSetArg(args[3], XmNeditMode,  XmMULTI_LINE_EDIT);
    text_wid = XmCreateScrolledText(main_w, "text_wid",
                                    args, 4);
    XtManageChild(text_wid);

    /*  read file and put data in text widget */
    read_file(text_wid);

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

void read_file(Widget  text_wid)

{
    static char *filename = "text.c";
    char  *text;
    struct stat statb;
    FILE *fp;

   /* check file is a regular text file and open it */

    if ( (stat(filename, &statb) == -1)
         || !(fp = fopen(filename, "r")))
    {   fprintf(stderr, "Cannot open file: %s\n", filename);
        XtFree(filename);
        return;
    }

    /* Map file text in the TextWidget */

    if (!(text = XtMalloc((unsigned)(statb.st_size+1))))
    {   fprintf(stderr, "Can't alloc enough space for %s",
                filename);
        XtFree(filename);
        fclose(fp);
        return;
    }

    if (!fread(text, sizeof(char), statb.st_size+1, fp))
        fprintf(stderr, "File read error\n");

    text[statb.st_size] = 0; /* be sure to NULL-terminate */

    /* insert file contents in TextWidget */

    XmTextSetString(text_wid, text);

    /* free memory
    */
    XtFree(text);
    XtFree(filename);
    fclose(fp);
}
```

```
void quit_call()

{   printf("Quitting program\n");
    exit(0);
}
```

44.4 Editing Text

Motif provides many functions that allow the editing of the text (`String`) stored in the widget. Text can be searched, inserted and replaced.

To replace all or parts of the text in a Text widget use the `XmTextReplace()` function. It has four arguments:

Widget — The Text widget.

Start Position (`XmTextPosition` type) — A `long int` measured in bytes. Specifies where to begin inserting text.

End Position (`XmTextPosition` type) — The end insertion point.

New text — The `String` that will replace existing text.

No matter how long the specified replacement text string is, text is *only* replaced (character-by-character) between the 2 positions. However, if the start and end positions are equal then text is inserted after the given position.

An alternative method to insert text, is to use the `XmTextInsert()` function. This takes 3 arguments:

Widget — the `Text` Widget,

Insert Position — the `XmTextPosition` for the insertion,

Insertion Text — the text `String`.

To Search for a string in the Text widget, use the `XmTextFindString()` function with the following arguments:

Text Widget — to be searched,

Start Position — as before,

Search `String`,

Search Direction — either `XmTEXT_FORWARD` or `XmTEXT_BACKWARD`,

Position — a `XmTextPosition` pointer that returns the position found.

`XmTextFindString()` returns a `Boolean` value which is `False` if no string was found.

To obtain text (in full) from a Text widget use the `XmTextGetString()` function to return a `String` for a specified widget. An example use of this function is to save text stored in the Text widget to a file. This can be simply achieved by:

- `XmTextGetString()` from the widget.
- Write the string (*e.g.* `fprintf()`) to a file.

The function `XmTextGetSubstring()` can be used to get a portion of text from a widget. It takes 5 arguments:

Text Widget — from where the substring is to be obtained.

Start Position — the `XmTextPostition` of the first character in the substring to be returned.

Number of Characters — to be copied from the substring.

Buffer Size — the number of characters in the `String` buffer where the substring is copied to.

String — a pointer to a `String` buffer which will hold the substring value when this function is returned.

Similar functions exist for both the TextField and the CSText widgets. An example of these functions in use with the TextField widget is given is Section 44.7.

44.5 Scrolling Control

Motif provides a variety of functions to control the scrolling of the text within a Text widget. Thus, the application programmer has control over which portions of text can be displayed at a given time. The following options are available:

- To show a piece of text in a given position, use `XmTextShowPostion()`.
- To set a line to be at the top of scrolled widget, specify the position to `XmTextSetTopCharacter()`.
- To position the cursor to a position where text may be inserted, use `XmTextSetInsertionPosition()`.

Similar functions exist for both the TextField and the CSText widgets. An example of these functions in use with the TextField widget is given is Section 44.7.

44.6 Text Callbacks

Behind almost all of the Text widget functions, described above, lie default callback resources. These control the basic text editing facilities: cut and paste, searching *etc.* However, there may be occasions when the application may need greater control over things. Several callback resources are provided for this purpose. Briefly theses are:

XmNactivateCallback — Called on **Enter** key press (only single-line Text widgets).

XmNverifyCallback — Used to verify a change to text *before* it is made.

XmNvalueChangedCallback — Called *after* a change to text has been made.

XmNmotionCallback — Called when the user has altered the cursor position, or made a text selection with mouse .

XmNfocusCallback — Called when the user wants to begin input.

XmNlosingfocusCallback — Called when the widget is losing keyboard focus.

We can use *verifyCallbacks* for checking user inputs — for example for password verification.

44.7 Editing and Scrolling in Practice

The `test_for_echo.c` program (Section 42.3) illustrates the use of some of the editing and scrolling functions described. The program basically operates as follows:

- Two TextField widgets are created that are then made the command and message areas of a MainWindow widget (Fig. 42.4).
- The command area Textfield is allowed to receive input but the message area is not. The `XmNeditable` resource is set to `False` for the message area widget, `msg_wid` in `test_for_echo.c`.
- After the widgets are instantiated the command and message area prompts are put into the respective widgets using the `XmTextFieldSetString()` function. The text insertion point is set for the command area so that input can be received after the command area prompt.
- When the command area receives a `XmNactivateCallback` event, a callback `cmd_cbk`, is called that parses the command area for a substring (excluding the command area prompt) using the `XmTextFieldGetSubstring()` function.
- The substring is then appended to the message area after the message area prompt using the `XmTextFieldReplace()`, with the start point set appropriately.
- The command area prompt is then reset by simply setting (using the `XmTextFieldSetString()` function) it to the original prompt value.
- The insertion point for the command area is also reset (using the `XmTextFieldSetInsertionPosition() function`) for subsequent data input.

44.8 Exercises

Exercise 44.1 *Modify the* `text.c` *(Section 44.3) so that it employs a File-Selection widget to allow the user to select a file, which it then reads and displays in a ScrolledText Widget .*

Chapter 45

List Widgets

The `List` widget allows selection from a variety of specified items. The `List` widget is actually one of the component widgets in the FileSelectionBox widget(Chapter 43).

The use of a List is similar to that of a Menu, but is a little more flexible:

- We can add and delete items from the list.
- Lists can be scrolled.
- A number of Selection modes are available.

An example of a List Widget is shown in Fig. 45.1.

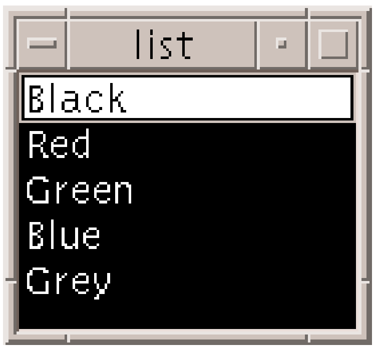


Figure 45.1: Output of `list.c`

45.1 List basics

To create a simple list use: `XtVaCreateManagedWidget()`, and specify `xmListWidgetClass` as the widget type (or use `XmCreateList()`. The header file `<Xm/List.h>` must be included. We will usually want to create a `ScrolledList`. To do this use: `XmCreateScrolledList()`.

There are a number of useful resources:

XmNitemCount — The number of items in the list.

XmNitems — The item list. The item list is a `XmStringTable` data type. This is basically a 1D array of XmStrings.

XmNselectionPolicy — Controls how items are chosen (*see* Section 45.1.1 below).

XmNvisibleItemCount — The number of items shown in the list. This determines height of widget.

XmNscrollBarDisplayPolicy — Either `XmAS_NEEDED` or `XmSTATIC`. `XmSTATIC` will always show (vertical) scroll even if all items are visible.

XmNlistSizePolicy — `XmCONSTANT, XmRESIZE_IF_POSSIBLE` or `XmVARIABLE`. Controls horizontal scrolling.

XmNselectedItemCount — The number of selected items.

XmNselectedItems — The selected items from the list. These can be set at list initialisation to enable a *default* choice to be specified.

45.1.1 List selection modes

One primary distinction between a list and menu is in the methods of selection available. Four types of selection are available. The `List` resource `XmNselectionPolicy` must be set accordingly:

Single — `XmSINGLE_SELECT` (defined in `<Xm/List.h>`). Only one item may be selected.

Browse — `XmBROWSE_SELECT`. Similar to Single selection, except a default selection can be provided and user interaction may vary.

Multiple — `XmMULTIPLE_SELECT`. More than one item may be selected. A mouse click on a item will select it. If you click on an already selected item it will be *deselected*.

Extended — `XmEXTENDED_SELECT`. Like Multiple selection. Here you can drag the mouse over a number of items to select them.

45.1.2 Adding and removing list items

As the name of this widget implies, the *List* is a dynamic structure that can grow or shrink as items are added or deleted.

To add an item to a list, use the function `XmListAddItem()`, which takes 3 arguments:

The List Widget — The widget we add items to,

Item — An (`XmString`) list item we wish to add,

Position — The (`int`) position. **Note:** List indexing **Starts at index 1** (Die hard C programmers take note). Position **0** in the list is used to specify the last position in the list. If you specify a **0** position items will get appended to the end of the List.

Another function `XmListAddItemUnselected()` has exactly the same syntax as `XmListAddItem()`, above. This function will guarantee that an item is not selected when it is added. This is not always the case with the `XmListAddItem()`, since selection will be dependent on the currently selected list index.

To remove a *single* named (`XmString`) item, `str`, from a `List` widget, use the
`XmListDeleteItem(Widget List, XmString str)` function.

To remove a number of named (`XmString`) items, use the `XmListDeleteItems(Widget List, XmString *del_items)` function where the second argument, `del_items`, is an array of `XmString`s that contain the names of items being deleted.

If you know the position of item(s) in a List, as opposed to their names, you can use the following functions:

`XmListDeletePos(Widget wid, int pos)` where the second `pos` argument specifies the deletion position.

`XmListDeleteItemsPos(Widget wid, int num, int pos)` where `num` items are deleted starting from position `pos`.

To delete *all* items form a list, use `XmListDeleteAllItems(Widget wid)`.

45.2 Selecting and Deselecting items

Two functions `XmListSelectItem(Widget, XmString, Boolean)` and `XmListSelectPos(Widget, int, Boolean)` may be used to select an item from within a program.

The Boolean value, if set to `True`, will call the callback function associated with the particular List.

To deselect items use `XmListDeselectItem(Widget, XmString)`, `XmListDeselectPos(Widget, int)` or `XmListDeselectAllItems(Widget)`. The operation of which is similar to corresponding delete functions.

45.2.1 List Enquiry

Since the List is *dynamic* we may need to know how long the list is, and which items are currently selected *etc.*. Some of these values can be obtained from the callback structure of a list (*see* Section45.3 below). However, if no callback has been invoked the programmer may sometimes still need to access this information.

The List resources are updated automatically (by default callback resources) so all we need to do is to `XtGetValues()` (or something similar) for the resource we want. For example:

- To find the length of a list:

  Obtain the value of the resource `XmNitemCount`.

```
Arg args[1]; /* Arg array */
int n = 1; /* number arguments */
int list_size; /* value to store XtGetValue() request */

......

XtSetArg(args[0], XmNitemCount, &list_size);
```

```
   XtGetValues(list_wid, args, n);

printf("The Size of the list = %d\n", list_size);
```

where `list_wid` is the List widget we request this information from.

Note: We pass the address of `list_size` to `XtSetArg()` since we need to specify a pointer to physical (program) memory in which to store the result. The value of `list_size` is available after the `XtGetValues()` call and is only *accurate* until the next user (or application) list addition/subtractions.

- To find the number of items currently selected:

  Obtain the value of the resource `XmNselectedItemCount`:

```
   Arg args[1]; /* Arg array */
   int n = 1; /* number arguments */
   int select_count; /* value to store
                        XtGetValue() request */

   ......

   XtSetArg(args[0], XmNselectedItemCount, &select_count);
   XtGetValues(list_wid, args, n);

printf("The Numver of selected list items = %d\n",
          select_count);
```

Recall that the use of `XtGetValue()` is similar to that of `XtSetValue()` (Chapter40).

45.3 List Callbacks

Default List callback functions facilitate common interaction with a List such as selection of an item (or multiple items) and addition or deletion of items. More importantly, related resource information is automatically updated (*e.g.* the current List size, `XmNitemCount`). There is a List callback resource for each of the selection types (*e.g.* `XmNsingleSelectionCallback`) and also a `XmNdefaultActionCallback`. The application programmer is free to add his own callback functions in the usual manner. In this case, the selection policy callback will be called first and then the default.

The Callback function has the usual form:

```
list_cbk(Widget w, XtPointer data, XmListCallbackStruct *cbk)
```

Elements of the `XmListCallbackStruct` include:

`item` — the XmString of the selection.

`item_position` — position in the List.

`selected_items` — the List of XmStrings in multiple selections.

`selected_item_count` — number of multiple selections.

`selected_item_positions` — positions in the List.

An example of the use of a List callback is given in the following `list.c` example program.

45.4 The `list.c` program

An example of a List in action is given in `list.c`.

We create a simple list that shows a selection of colours. Selection of these colours changes the background colour[1] of the List widget.

```
#include <Xm/Xm.h>
#include <Xm/List.h>


/* Prototype Callback */
```

[1]In order to illustrate the list example with a non-trivial and vaguely interesting application we actually use some Xlib colour routines. These have not yet been formally addressed (*see* Chapter 51). However, the use of colour routines here are not too difficult to comprehend and basically resort to the list callback `list_cbk()` setting an appropriate `XmNbackground` resource value. The reader is urged to concentrate on the List creation and usage matters in this example.

```
void list_cbk(Widget , XtPointer ,
              XmListCallbackStruct *);

String colours[] = { "Black",  "Red", "Green",
                     "Blue", "Grey"};

Display *display; /* xlib id of display */
Colormap cmap;

main(int argc, char *argv[])

{
    Widget          top_wid, list;
    XtAppContext    app;
    int             i, n = XtNumber(colours);
    XColor          back, fore, spare;
    XmStringTable   str_list;
    Arg             args[4];


    top_wid = XtVaAppInitialize(&app, "List_top", NULL, 0,
        &argc, argv, NULL, NULL);

    str_list =
      (XmStringTable) XtMalloc(n * sizeof (XmString *));

    for (i = 0; i < n; i++)
        str_list[i] = XmStringCreateSimple(colours[i]);

    list = XtVaCreateManagedWidget("List",
        xmListWidgetClass,     top_wid,
        XmNvisibleItemCount,   n,
        XmNitemCount,          n,
        XmNitems,              str_list,
        XmNwidth,               300,
        XmNheight,              300,
        NULL);
    for (i = 0; i < n; i++)
        XmStringFree(str_list[i]);
    XtFree(str_list);

    /* background pixel to black foreground to white */

    cmap = DefaultColormapOfScreen(XtScreen(list));
    display = XtDisplay(list);

    XAllocNamedColor(display, cmap, colours[0], &back,
                     &spare);
    XAllocNamedColor(display, cmap, "white", &fore, &spare);

    n = 0;
    XtSetArg(args[n],XmNbackground, back.pixel);
    ++n;
    XtSetArg(args[n],XmNforeground, fore.pixel);
    ++n;
    XtSetValues(list, args, n);

    XtAddCallback(list, XmNdefaultActionCallback, list_cbk,
                  NULL);

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

/* called from any of the "Colour" list items.
   Change the color of the list widget.
   Note: we have to use dynamic setting with XtSetValues()..
 */
void list_cbk(Widget w, XtPointer data,
         XmListCallbackStruct *list_cbs)

{   int n =0;
    Arg args[1];
    String selection;
    XColor xcolour, spare; /* xlib color struct */

    /* list->cbs holds XmString of selected list item */
    /* map this to "ordinary" string */

    XmStringGetLtoR(list_cbs->item, XmSTRING_DEFAULT_CHARSET,
                    &selection);

    if (XAllocNamedColor(display, cmap, selection,
                         &xcolour, &spare) == 0)
       return;

    XtSetArg(args[n],XmNbackground, xcolour.pixel);
    ++n;
    /* w id of list widget passed in */
    XtSetValues(w, args, n);

}
```

Chapter 46

The Scale Widget

The Scale widget allows the user to input numeric values into a program. An example of the Scale widget is shown in Fig. 46.1.

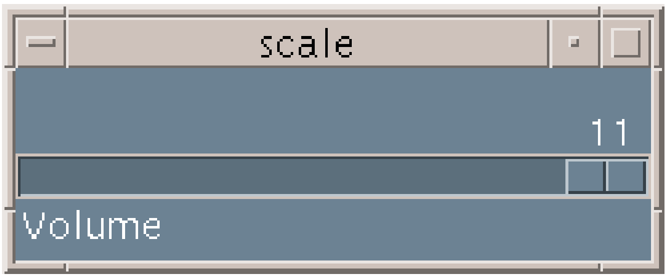


Figure 46.1: Output of `scale.c`

46.1 Scale basics

To create a Scale Widget use `XtVaCreateManagedWidget()`, with a `xmScaleWidgetClass` class pointer or use `XmCreateScale()`. Once again a header file, `<Xm/Scale.h>`, needs to be included in all programs using Scale widgets.

The following Scale Widget Resources are typically used:

XmNmaximum — The largest value of scale.

XmNminimum — The scale's smallest value. The default value is 0.

XmNorientation — `XmHORIZONTAL` or `XmVERTICAL`, resource values define how the Scale is displayed.

XmNtitleString — The `XmString` label of the scale.

XmNdecimalPoints — A scale value is always returned as an integer (`default` 0). This resource can be set to give the user the impression of floating point input, *e.g.* if we have a range 0 – 1000 on the Scale but `XmNdecimalPoints` set to 2. The displayed range would be 0.00 – 10.00.

The application programmer must take care of the input value to the program. In the above a value will still get returned in the integer range and somewhere in the application there must be a division by 100.

XmNshowValue — `True` or `False`. This resource decides whether or not to display the value of the scale as it moves.

XmNvalue — The current value (`int`) of the Scale.

XmNprocessingDirection — Either `XmMAX_ON_TOP`, `XmMAX_ON_BOTTOM`, `XmMAX_ON_LEFT` or `XmMAX_ON_RIGHT`. This resource sets the end of the scale, where the maximum and minimum values are placed. This depends on the orientation of the Scale.

46.2 Scale Callbacks

The Scale callback can be called for two types of events:

XmNvalueChangedCallback — If the value is changed.

XmNdragCallback — If the Scale value is moved at all, "continuous" values can be input. **Note:** This may affect X performance as it involves *instantaneously* updating the display of the scale.

The Scale callback function is standard:

```
void scale_cbk(Widget w, XtPointer data,
               XmScaleCallbackStruct *struct)
```

The structure element `value` holds the current Scale **integer** value, and is the only structure element that is really of interest. The `scale.c` program illustrates this usage.

46.3 The scale.c program

The program simply brings up a virtual *volume* Scale (in the context of a virtual amplifier controller). The user changes the value which is caught by a `XmNvalueChangedCallback` and the current `value` is interrogated to print a message to standard output.

```
#include  <Xm/Xm.h>
#include <Xm/Scale.h>

/* Prototype callback  */

void scale_cbk(Widget , int ,
               XmScaleCallbackStruct *);

main(int argc, char **argv)

{   Widget        top_wid, scale;
    XmString   title;
    XtAppContext  app;


    top_wid = XtVaAppInitialize(&app, "Scale_eg", NULL, 0,
        &argc, argv, NULL, NULL);

    title = XmStringCreateLocalized("Volume");

    scale = XtVaCreateManagedWidget("scale",
        xmScaleWidgetClass, top_wid,
        XmNtitleString,   title,
        XmNorientation,    XmHORIZONTAL,
        XmNmaximum,       11,
        XmNdecimalPoints, 0,
        XmNshowValue,     True,
        XmNwidth,         200,
        XmNheight,        100,
        NULL);

    XtAddCallback(scale,XmNvalueChangedCallback, scale_cbk,
                  NULL);

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}


void scale_cbk(Widget widget, int data,
               XmScaleCallbackStruct *scale_struct)

{    if (scale_struct->value < 4)
       printf("Volume too quiet (%d)\n");
    else if (scale_struct->value < 7)
       printf("Volume Ok (%d)\n");

     else
        if (scale_struct->value < 10)
       printf("Volume too loud (%d)\n");
      else /* Volume == 11 */
       printf("Volume VERY Loud (%d)\n");
}
```

46.4 Scale Style

46.4.1 Motif 1.2 Style

The *Motif Style Guide*(Chapter 52) suggests that some sort of markers should be used to gauge distance along a Scale. However, no provision is made for this within the Scale widget class in Motif 1.2. Instead, the programmer must assemble this manually. An assortment of labels and tics can be used to provide some sort of visual *ruler* (Fig. 46.2).

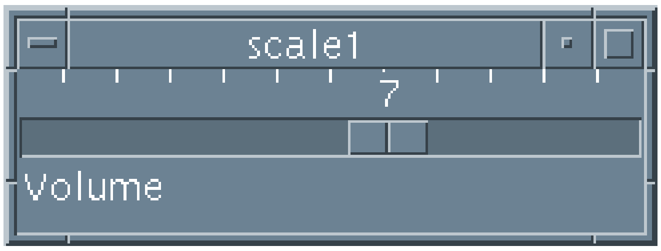


Figure 46.2: Better Style Scale Output

The Motif program code that achieves this, by placing vertical SeparatorGadgets ("|") at equally spaced intervals, is as follows:

```
Widget ...,tics[1];

.......

.......

/* label scale axis */
    for (i=0; i < 11; ++i )
      {  XtSetArg(args[0], XmNseparatorType, XmSINGLE_LINE);
         XtSetArg(args[1], XmNorientation, XmVERTICAL);
         XtSetArg(args[2], XmNwidth, 10);
         XtSetArg(args[3], XmNheight, 5);
         tics[i] = XmCreateSeparatorGadget(scale, "|",
                 args, 4);
      }

    XtManageChildren(tics, 11);

...........
...........
```

46.4.2 The Motif 2.0 Style

Motif 2.O provides a new convenience function `XmScaleSetTicks()` to allow for an easier configuration of ticks along the Scale widget. The configuration allows for three different sized ticks to be placed at specified regular intervals along the Scale. Each tick mark is actually a SeparatorGadget oriented perpendicular to the Scale's orientation. The function `XmScaleSetTicks()` takes seven arguments:

Widget — The scale widget being assigned the ticks.

int `large_every` — The number of Scale values between large ticks.

int `num_medium` — The number of medium ticks between large ticks.

int `num_small` — The number of small ticks between medium ticks.

Dimension `size_large` — The size (width or height) of the big ticks.

Dimension `size_medium` — The size (width or height) of the medium ticks.

Dimension `size_small` — The size (width or height) of the small ticks.

If you specify tick marks for a Scale and then change the Scale's orientation then you must remove all the tick marks and then recreate new ones in the correct orientation. This may be achieved by the following method:

- To remove ticks marks you must destroy all SeparatorGadget tick mark children:
  - The first two children of a Scale are its title and scroll bar.
  - *All* additional children are the tick marks.
  - The function `XtDestroyChildren()` can be used to remove specified tick marks.
- Call `XmScaleSetTicks()` with appropriate arguments.

Chapter 47

ScrolledWindow and ScrollBar Widgets

We have already seen scrollbars in action with Text (Chapter 44) and List widgets (Chapter 45). More generally, Motif provides a ScrolledWindow widget that allows scrolling of any widget contained within it. This means that we can place a large view area inside a smaller one and then view portions of the view area.

As we have seen, Motif actually provides convenience functions to produce ready made ScrolledText and ScrolledList widgets. These are, in fact, Text or List widgets contained inside a ScrolledWindow widget.

ScrollBars are the basic components of scrolling. A ScrolledWindow widget may contain either, or both of, horizontal and vertical ScrollBars. Many application will typically use ScrollBars. In the majority of instances, ScrollBar widgets will be created automatically by higher level widgets (*e.g.* ScrolledWindow, ScrolledText or ScrolledList). Occasionally, greater control over the default settings of the ScrollBar will be required. Sometimes, these resources can be set as resources of the higher level widget, other times the resources will need to be set explicitly. In this chapter, we will highlight important ScrollBar resources and illustrate how they can be set in both of the above scenarios.

47.1 ScrolledWindow Widgets

ScrolledWindow widgets can be created manually with `XtVaCreateManagedWidget()`, with the `xmScrolledWindowWidgetClass` pointer or with `XmCreateScrolledWindow()` function.

Associated definitions for this widget class *etc.* are included in the `<Xm/ScrolledW.h>` header file.

Useful resources include:

XmNhorizontal, XmNvertical — The identifiers (IDs) of component ScrollBar widgets of the ScrolledWindow. One or both may be present.

XmNscrollingPolicy — Either defined as `XmAUTOMATIC` or `XmAPPLICATION_DEFINED`. If the scrolling policy is `XmAUTOMATIC` then scrolling is taken care of otherwise the application must create and control XmScrollBar widgets itself.

XmNscrolBarDisplayPolicy — Either defined as `XmAS_NEEDED` or `XmSTATIC`, as with List Scrolling.

XmNvisualPolicy — Either defined as `XmCONSTANT` or `XmVARIABLE`. If constant then scrolled window cannot resize itself.

XmNworkWindow — The widget to be scrolled.

47.2 ScrollBar Widgets

You may have to create ScrollBars yourself, especially if the scrolling policy is defined as `XmAPPLICATION_DEFINED`. Alternatively, you may get the ID of a ScrollBar from a ScrolledWindow (*e.g.* `XmNhorizontal` resource).

To create a ScrollBar, use `XtVaCreateManagedWidget()` with `xmScrollBarWidgetClass` or use `XmCreateScrollBar()`.

To obtain a horizontal ScrollBar ID from a ScrolledWindow:

```
Arg args[1]; /* Arg array */
int n = 1; /* number arguments */
Widget scrollwin,scrollbar;
```

```
scrollwin = XmCreateScrolledWindow(.....)
......

XtSetArg(args[0], XmNhorizontal, &scrollbar);
XtGetValues(scrollwin, args, n);
```

Typical ScrollBar resources include:

XmNsliderSize — A slider may be divided up into *unit lengths*. This resource sets its size.

XmNmaximum — The largest size (measured in unit lengths) a ScrollBar can have.

XmNminimum — The smallest ScrollBar size.

XmNincrement — The number of unit lengths the Scale will change when moved with mouse.

XmNorientation — `XmVERTICAL` (Default) or `XmHORIZONTAL` layout of the widget.

XmNvalue — The current position of the Scale.

XmNpageIncrement — Controls how much the underlying *work window* moves relative to a ScrollBar movement.

The Callback resources for a ScrollBar are:

XmNvalueChangedCallback — Called if the ScrollBar value changes.

XmNdecrementCallback, XmNincrementCallBack — Called if the ScrollBar value changes up or down.

XmNdragCallback — Called on *continuous* Scale value updates.

XmNpageDecrementCallback, XmNpageIncrementCallback — Called on movement of *work window*.

XmNtoTopCallback, XmNtoBottomCallback — Called if ScrollBar is moved to minimum or maximum values.

Chapter 48

Toggle Widgets

A Toggle Widget basically provides a simple switch type of selection. The Toggle is either a square or circle shape indicator which if pressed can be turned on and if pressed again turned off. Text and pictorial items (*pixmaps*) can be used to label a Toggle.

Several Toggle widgets can be grouped together to allow greater control of selection. Motif provides *two* methods of grouping Toggles together.

RadioBox Widget — Only one of the group can be *on* at any given time. Toggles are diamond (Motif 1.2) or circular (CDE 1.0 and Motif 2.0) in this widget (Fig 48.1).

CheckBox Widget — More than one Toggle can be *on* at any time. Toggles are square. (Fig. 48.2).

48.1 Toggle Basics

To create a single Toggle use `XtVaCreateManagedWidget()` with a `xmToggleButtonWidgetClass` pointer or use `XmCreateToggleButton()`.

The header file `<Xm/ToggleB.h>` holds definitions *etc* for this widget.

Several Toggle Resources are relevant to the programmer:

XmNindicatorType — Defines the mode of operation of the Toggle. Set this resource to `XmN_OF_MANY` for a CheckBox or `XmONE_OF_MANY` for a RadioBox.

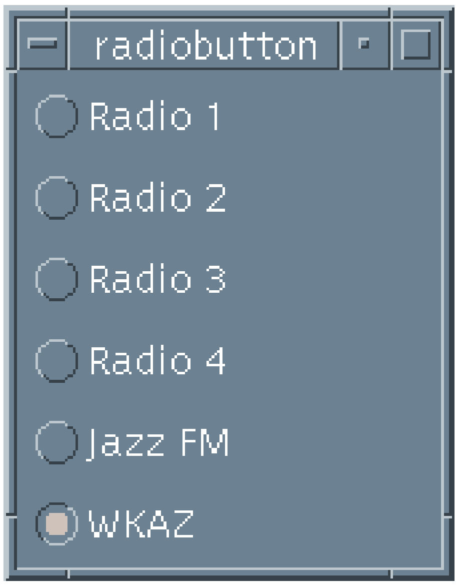


Figure 48.1: A RadioBox set of Toggle Widgets

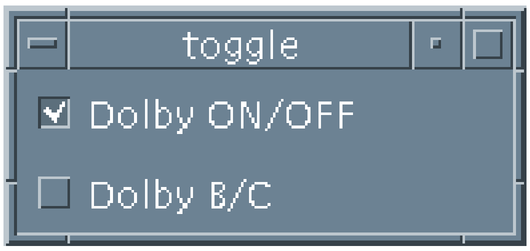


Figure 48.2: A CheckBox set of Toggle Widgets

XmNindicatorOn — Set this resource to `True` to turn indicator on. The *default* is `False` (off).

XmNindicatorSize — Changes indicator size. Size is specified in Pixel unit size.

XmNlabelType — Either set to `XmSTRING` or `XmPIXMAP`. Defines the type of label associated with the Toggle.

XmNlabelString — The `XmString` label of Toggle.

XmNlabelPixmap — The Pixmap of an unselected Toggle.

XmNselectPixmap — The Pixmap of a selected Toggle.

XmNselectColour — The colour of a selected Toggle. (`Pixel` data type).

The `Pixmap` is a standard X data type. You can use `XmGetPixmap()` to load in a `Pixmap` from a file. (See Chapter 49 on Graphics and Xlib for further details.)

48.2 Toggle Callbacks

Toggle Callbacks have the usual callback function format and are prototyped by:

```
void toggle_cbk(Widget, XtPointer,
                XmToggleCallbackStruct *).
```

Toggle Callbacks are activated upon an `XmNvalueChangedCallback`. The `XmToggleCallbackStruct` has a boolean element `set` that is `True` if the Toggle is **on**. The `toggle.c` (Section 48.3 below) illustrates the use of Toggle callbacks.

48.3 The `toggle.c` program

This program creates two CheckBox Toggles in a RowColumn Widget. When their Callbacks are activated, the state of the Toggle is interrogated and a message is printed to standard output.

```
#include <Xm/Xm.h>
#include <Xm/ToggleB.h>
#include <Xm/RowColumn.h>

/* Prototype callback */
void toggle1_cbk(Widget , XtPointer ,
XmToggleButtonCallbackStruct *),
void toggle2_cbk(Widget , XtPointer ,
XmToggleButtonCallbackStruct *);

main(int argc, char **argv)

{
    Widget toplevel, rowcol, toggle1, toggle2;
    XtAppContext app;


    toplevel = XtVaAppInitialize(&app, "Toggle", NULL, 0,
        &argc, argv, NULL, NULL);

    rowcol = XtVaCreateWidget("rowcol",
        xmRowColumnWidgetClass, toplevel,
        XmNwidth, 300,
        XmNheight, 200,
        NULL);

    toggle1 = XtVaCreateManagedWidget("Dolby ON/OFF",
        xmToggleButtonWidgetClass, rowcol, NULL);

    XtAddCallback(toggle1, XmNvalueChangedCallback,
              toggle1_cbk, NULL);

    toggle2 = XtVaCreateManagedWidget("Dolby B/C",
            xmToggleButtonWidgetClass, rowcol, NULL);

    XtAddCallback(toggle2, XmNvalueChangedCallback,
              toggle2_cbk, NULL);
    XtManageChild(rowcol);
    XtRealizeWidget(toplevel);
    XtAppMainLoop(app);
}

void toggle1_cbk(Widget widget, XtPointer client_data,
XmToggleButtonCallbackStruct *state)

{  printf("%s: %s\n", XtName(widget),
            state->set? "on" : "off");
}


void
toggle2_cbk(Widget widget, XtPointer client_data,
XmToggleButtonCallbackStruct *state)

{ printf("%s: %s\n", XtName(widget), state->set ? "B" : "C");
}
```

48.4 Grouping Toggles

You can, if you wish, group RadioBoxes or CheckBox Toggles in RowColumn, or Forms yourself (as has been done in the above `toggle.c` program).

However, Motif provides a few convenience functions, `XmCreateSimpleRadioBox()` and `XmCreateSimpleCheckBox()` are common examples.

Basically, these are RowColumn Widgets with Toggle children created automatically. Appropriate resources can be set, *e.g.* XmNindicatorType or XmNradioBehaviour (in this enhanced RowColumn Widget) .

Chapter 49

Xlib and Motif

This Chapter will deal specifically with the introduction of **Xlib** – *the low level X library*. Recall that Xlib provides the means of communication between the application and the X system. The Xlib library of (C) subroutines is large and of a comparable size to Motif. Many of Xlib's routines deal with the creation, maintenance and interaction between windows and applications. Xlib does not have any concept of widgets and thus does not provide provide any (high level) means of interaction. In general, writing complete application solely in Xlib is not a good idea. Motif provides many useful, complete GUI building blocks that should always be used if available. For example, do you really want to write a complete text editing library in Xlib, when Motif provides one for free?

If you use Motif then there should never be any need to resort to Xlib for window creation. Motif is far more powerful and flexible. Consequently, in this and forthcoming Chapters, we will only deal with issues that affect the interfacing of Xlib with Motif and the Xt toolkit.

However, for certain tasks we will have to resort to Xlib. The sort of tasks that we will be concerned with in this text are:

Graphics — The responsibility of actually drawing items to a window is the domain of Xlib. Whilst Motif does provide a *canvas* where graphics can be drawn, Motif has to rely on Xlib functions to actually draw something. Xlib only facilitates simple 2D graphics.

Pixmaps — A Pixmap is an off-screen drawable area where graphics may be placed. Pixmaps are clearly related to usage with Xlib graphics.

However, Pixmaps are also used with Images and also to *label* graphics based widgets *e.g. DrawnButton and Toggle Widgets.*

Colour — Handling colour is a fundamental element for a windowing system. Colour plays an important part in any GUI for providing a user friendly GUI layout, indicating key features and alerting the user to certain actions. However, in the X system colour can be quite complex as a variety of devices are required to be supported by the X network and each device may have a completely different interpretation of colour. Because of this requirement, colour needs to be handled at the Xlib level. Chapter 51 deals with the major issues of colour and X.

Text — Just like colour handling of Text can vary across devices. The font, typeface (*e.g.* italic, bold) and size may need to be controlled by Xlib.

Events — Whilst Motif handles events in a robust and efficient manner, the Motif method of event handling is sometimes a little restrictive. In this case responsibility for event handling is sometimes handed over to Xlib.

49.1 Xlib Basics

In Chapter 36 we described the X Window system and explained the relevance of each system component. We briefly mentioned that Xlib provides the interface between the X Protocol and the application program. Xlib therefore has to deal with many low level tasks. In short, Xlib concerns itself with:

- Display and Server Communication,
- Event and Error handling,
- Window Management — communication with the window manager,
- Text — fonts, size style *etc.*,
- Graphics — 2D graphics routines,
- Colour.

Xlib deals with much lower level objects than widgets. When you write or draw in Xlib reference, may be made to the following:

Display — This structure is used by nearly all Xlib functions. It contains details about the server and its screens.

Drawable — this is an identifier to a structure that can contain graphics (*i.e.* it can be **drawn** to). There are two common types:

> **Window** — A part of the screen.
>
> **Pixmap** — An *off-screen* store of graphical data.

Screen — Information about a *single* screen of a display.

Graphics Context (GC) — When we draw we need to specify things like line width, line style (*e.g* solid or dashed), colour, shading or filling patterns *etc.*. In Xlib we pass such information to a drawing routine via a GC structure. This must be created (and values set) before the routine uses the GC. More than one GC can be used.

Depth — the number of bits per pixel used to display data.

Visual — This structure determines how a display handles screen output such as colour and depends on the number of bits per pixel.

If we are programming in Xlib alone, we would have to create windows and open displays ourselves. However, if we are using a higher level toolkit such a Motif and require to call on Xlib routines then we need to obtain the above information from an appropriate widget in order pass on appropriate parameter values in the Xlib function calls.

Functions are available to obtain this information readily from a Widget. For example `XtDisplay(), XtWindow(), XtScreen()` *etc.* can be used to obtain the ID of a given Xlib `Display`, `Window`, or `Screen` structure respectively from a given widget. *Default* Values of these structures are also typically used. Functions `DefaultDepthofScreen(), RootWindowofScreen()` are examples.

Sometimes, in a Motif program, you may have to create an Xlib structure from scratch. The GC is the most frequently created structure that concerns us. The Function `XCreateGC()` creates a new GC data structure.

We will look at the mechanics of assembling Xlib graphics within a Motif program when we study DrawingAreas in Chapter 50. For the remainder of this Chapter we will continue to introduce basic Xlib concepts. In the coming Sections, reference is made to programs that are described in Chapter 50.

49.2 Graphics Contexts

As mentioned in the previous Section, the GC is responsible for setting the properties of lines and basic (2D) shapes. GCs are therefore used with every Xlib drawing function. The `draw.c` (Section 50.3.1) program illustrates the setting of a variety of GC elements.

To create a GC use the (Xlib) function `XCreateGC()`. It has 4 parameters:

- the `Display` ID (pointer),
- the `Drawable` ID,
- **mask** (`unsigned long`) — this controls how members of the following `XGCValues` structure may be set.
- a pointer to a `XGCValues` structure. This structure contains elements that can be set to change the values of an existing GC.

The `XGCValues` structure contains elements like `foreground, background, line_width, line_style`, *etc.* that we can set for obvious results. The `mask` has predefined values such as `GCForeground, GCBackground`, and `GCLineStyle`.

In `draw.c` we create a GC structure, `gc`, and set the foreground.

Xlib provides two macros `BlackPixel()` and `WhitePixel()` which will find the default black and white pixel values for a given `Display` and `Screen` if the default colourmaps are installed. Note that the reference to `BlackPixel()` and `WhitePixel()` can be a little confusing since the pixel colours returned may not necessarily be Black or White. `BlackPixel()` actually refers to the foreground colour and `WhitePixel()` refers to the background colour.

Therefore, to create a GC that *only* sets foreground colour to the default for a given `display` and `screen`:

```
gcv.foreground = BlackPixel(display, screen);
gc = XCreateGC(display, screen, GCForeground, &gcv);
```

where `gcv` is a XCGValues structure and `gc` a GC structure and `GCForeground` sets the `mask` to only allow alteration of the foreground.

To set *both* background and foreground:

```
gcv.foreground = BlackPixel(display, screen);
gcv.background = WhitePixel(display, screen);
gc = XCreateGC(display, screen,
               GCForeground | GCBackground, &gcv);
```

where we use the `|` (OR) in the `mask` parameter that allows <u>both</u> the values to be set in the XGCValues structure.

An alternative way to change GC elements is to use Xlib convenience functions to set appropriate GC values. Example functions include :

`XSetForeground(), XSetBackground(), XSetLineAttributes()`.
These set GC values for a given `display` and `gc`, for example:

```
XSetBackground(display, gc, WhitePixel(display, screen));
```

Further examples of their use are shown in the `draw.c` program (Section 50.3.1).

49.3 Two Dimensional Graphics

Xlib provides a whole range of 2D Graphics functions. See `draw.c` for examples in use. Most of these functions are fairly easy to understand and use. The functions basically draw a specific graphical primitive (point, line, polygon, arc, circle *etc.*) to a display according to a specific GC.

The simplest function is `XDrawPoint(Display *d, Drawable dr, GC gc, int x, int y)` which draws a point at position `(x, y)` to a given `Drawable` on a `Display`. This effectively colours a single pixel on the Display.

The function `XDrawPoints(Display *d, Drawable dr, GC gc, XPoint *pts, int n, int mode)` is similar except that an `n` element array of `XPoints` is drawn. The `mode` may be defined as being either `CoordModeOrigin` or `CoordModePrevious`. The former mode draws all points relative to the origin whilst the latter mode draws relative to the last point.

Other Xlib common drawing functions include:

`XDrawLine(Display *d, Drawable dr, GC gc, int x1, int y1, int x2, int y2)` draws a line between `(x1, y1)` and `(x2, y2)`.

`XDrawLines(Display *d, Drawable dr, GC gc, XPoint *pts, int n, int mode)` draws a series of connected lines — taking pairs of coordinates in the list. The `mode` is defined as for the `XDrawPoints()` function.

`XDrawRectangle(Display *d, Drawable dr, GC gc, int x, int y, int width, height)` draws a rectangle with top left hand corner coordinate `(x, y)` and of `width` and `height`.

`XFillRectangle(Display *d, Drawable dr, GC gc, int x, int y, int width, int height)` fills (shades interior) a rectangle. The `fill_style` controls what shading takes place.

`XFillPolygon(Display *d, Drawable dr, GC gc, XPoint *pts, int n, int mode, int mode)` fills a polygon whose outline is defined by the `*pts` array.

This function behaves much like `XDrawLines()`.

The `shape` parameter is either `Complex, Nonconvex` or `Convex` and controls how the server may configure the shading operation.

`XDrawArc(Display *d, Drawable dr, GC gc, int x, int y, int width, int height, int angle1, int angle2)` draws an arc.

`XFillArc(Display *d, Drawable dr, GC gc, int x, int y, int width, int height, int angle1, int angle2)` draws an arc and fills it.

The `x, y, width` and `height` define a *bounding* box for the arc. The arc is drawn(Fig. 49.1) from the centre of the box. The `angle1` and `angle2` define the start and end points of the arc. The `angles` specify 1/64th degree steps measured anticlockwise. The `angle1` is relative to the 3 o'clock position and `angle2` is relative to `angle1`.

Thus to draw a whole circle set the `width` and `height` equal to the diameter of the circle and `angle1` = 0, `angle2` = 360*64 = 23040.

49.4 Pixmaps

If a window has been obscured then we will have to redraw the window when it gets re-exposed. This is the responsibility of the application and **not** the X window manager or system. In order to redraw a window we may have to

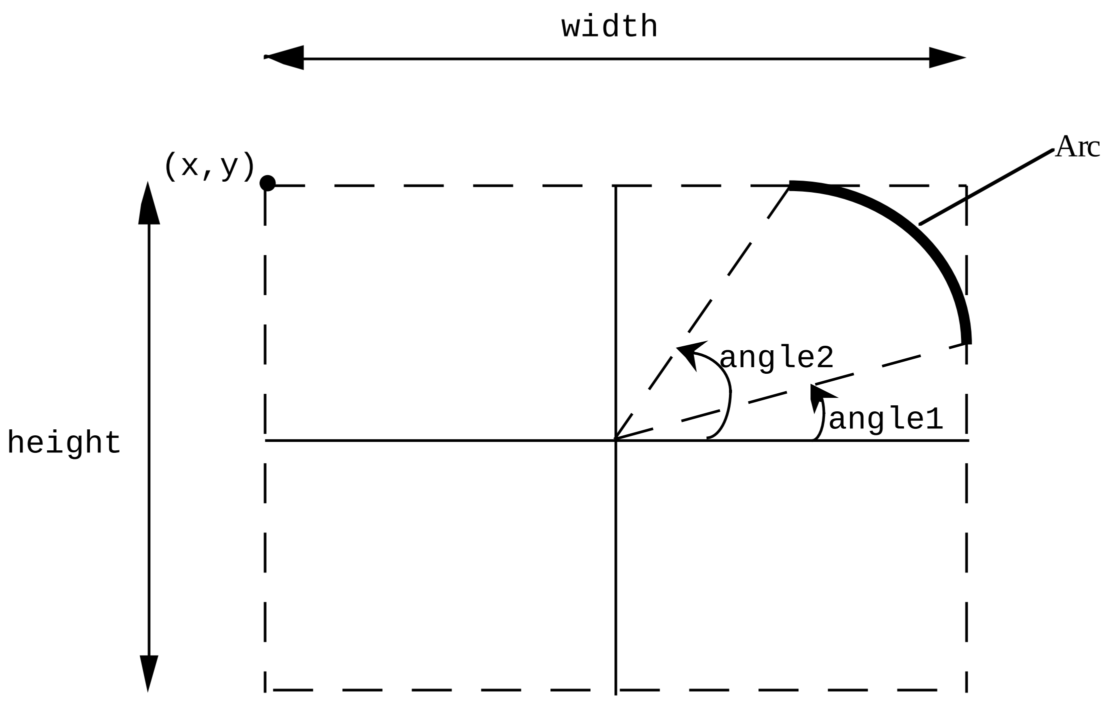


Figure 49.1: Drawing an Arc

go through all the drawing function calls that have previously been used to render our window's display. However, re-rendering a display in this manner will be cumbersome and may involve some complicated storage methods — there maybe be many drawing functions involved and the order in which items are drawn may also be important

Fortunately, X provides a mechanism that overcomes these (and other less serious) problems. The use of Xlib **Pixmaps** is the best approach.

A *Pixmap* is an off-screen *Drawable* area.

We can draw to a `Pixmap` in the same way as we can draw to a Window. We use the standard Xlib graphics drawing functions (Section 49.3) but instead of specifying a `Window` ID as the `Drawable` we provide a `Pixmap` ID. Note, however, that no immediate visual display effect will occur when drawing to a `Pixmap`. In order to see any effect we must *copy* the Pixmap to a Window.

The program `draw_input2.c` (Section 50.3.3) draws to pixmaps instead of to the window.

To create a Pixmap the `XCreatePixmap()` function should be used. This function takes 5 arguments and returns a *Pixmap* structure:

Display* — A pointer to the `Display` associated with Pixmap.

Drawable — The screen on which to place Pixmap.

Width, Height — The dimensions of the Pixmap.

Depth — The number of bits of each *pixel*. Usually 8 by default. A Pixmap of depth 1 is usually called a *bitmap* and are used for icons and special *bitmap files*.

When you have finished using a Pixmap it is a good idea to *free* the memory in which it has been stored by calling:

```
XFreePixmap(Display*, Pixmap).
```

If you want to clear a Pixmap (not done automatically) use `XFillRectangle()` to draw a background coloured rectangle that is the dimension of the whole Pixmap or use XClearArea(), XClearWindow() or similar functions.

To copy a Pixmap onto another Pixmap or a Window use:

```
XCopyArea(Display *display, Drawable source,
     Drawable destination, GC gc,
     int src_x, src_y,
     int width, int height,
     int dest_x, int dest_y);
```

where (`src_x, src_y`) specify the coordinates in the source pixmap where copy starts, `width` and `height` specify the dimensions of the copied area and (`dest_x, dest_y`) are the start coordinates in the destination where pixels are placed.

49.5 Fonts

Fonts are necessary in Motif as all `XmString` are drawn to the screen using fonts residing in the X system. A *font* is a complete set of characters (upper-case and lower-case letters, punctuation marks and numerals) of one size and one typeface. In order for a Motif program to gain access to different typefaces, fonts must be loaded onto the X server. All X fonts are bitmapped.

Not all X servers support all fonts. Therefore it is best to check if a specific font has been loaded correctly within your Motif program. There is a standard X application program, *xlsfonts*, that lists the fonts available on a particular workstation.

Each font or character set name is referred to by a `String` name. Fonts are loaded into to an Xlib `Font` structure using the `XLoadFont()` function with a given `Display` ID and font name argument. The function returns a `Font` structure.

Another similar function is `XLoadQueryFont()` which takes the same arguments as above but returns an `XFontStruct` which contains the `Font` structure plus information describing the font.

An example function, `load_font()` which loads a font named `‘‘fixed’’`, which should be available on most systems but is still checked for is given below:

```
void load_font(XFontStruct **font_info)

{  Display *display;
   char *fontname = "fixed";
   XFontStruct *font_info;

   display = XtDisplay(some_widget);
   /* load and get font info structure  */

  if (( *font_info = XLoadQueryFont(display, fontname)) == NULL)
      { /* error - quit early */
         printf("Cannot load  %s font\n",  fontname);
         exit(1);
      }
}
```

Motif actually possesses its own font loading and setting functions. These include `XmFontListCreate()`, `XmFontListEntryCreate()`, `XmFontListEntryLoad()` and `XmFontListAdd()`. These are used in a similar fashion to the Xlib functions above except that they return an `XmFontList` structure. However, in this book, we will be only using fonts at the Xlib level and these Motif functions will not be considered further.

49.6 XEvents

We should now be familiar with the basic notion of events in X and Motif. Mouse button presses, mouse motion and keyboard presses can be used to action menu, buttons *etc.*. These are all instances of events. Usually we are happy to let Motif take care of event scheduling with the `XtAppMainLoop()` function, and the setting of appropriate callback resources for widgets (Chapter 38).

Sometimes we may need to gain more control of events in X. To do this we will need to resort to Xlib. A specific example of this will be met in the Chapter 50 (the `draw_input1.c` program), where the default interaction, provided via callbacks in Motif, is inadequate for our required form of interaction.

49.6.1 XEvent Types

There are many types of events in Xlib. A special `XEvent` structure is defined to take care of this.

XEvents exist for all kinds of events, including: mouse button presses, mouse motions, key presses and events concerned with the window management. Most of the mouse/keyboard events are self explanatory and we have already studied them a little. Let us look at some window events further:

XConfigureNotify — If a window changes size then this event is generated.

XCirculateNotify — The event generated if the stacking order of windows has changed.

XColormapNotify — The event generated if colormap changes are made.

XCreateNotify, XDestroyNotify — The events generated when a window is created or deleted respectively.

XExpose, XNoExpose — Windows can be stacked, moved around *etc.* If part of a window that has been previously obscured becomes visible again then it will need to be redrawn. An `XExpose` event is sent for this purpose. An `XExpose` event is also sent when a window first becomes visible.

Note: There is no guarantee that what has previously been drawn to the window will become immediately visible. In fact, it is totally up to the programmer to make sure that this happens by picking up an `XExpose` event (See Sections 49.4 and 50.3.3 on Pixmaps and DrawingAreas).

49.6.2 Writing Your Own Event Handler

Most Motif applications will not need to do this since they can happily run within the standard application main loop event handling model. If you do need to resort to creating your own (Xlib) event handling routines, be warned: it can quickly become complex, involving a lot of Xlib programming.

Since, for the level of Motif programming described in this text, we will not need to resort to writing elaborate event handlers ourselves we will only study the basics of Motif/Xlib event handling and interaction.

The first step along this path is attaching a callback to an `XEvent` rather than a Widget callback action. From Motif (or Xt) you attach a callback to a particular event with the function `XtAddEventHandler()`, which takes 5 parameters:

Widget — the ID of the widget concerned.

EventMask — This can be used to allow the widget to be receptive to specific events. A complete list of event masks is given in the online support reference material. Multiple events can be assigned by ORing (`|`) masks together.

Nonmaskable — A `Boolean` almost always set to `False`. If it is set to `True` then it can be activated on *nomaskable* events such as `ClientMessage, Graphics Expose, Mapping Notify, NoExpose, SelectionClear, SelectionNotify or SelectionRequest`.

Callback — the callback function.

Client Data — Additional data to be passed to the event handler.

As an example we could set an `expose_callbck()` to be called by an `Expose` event by the following function call:

```
XtAddEventHandler(widget, ExposureMask, False,
                  expose_callbck, NULL);
```

To set a callback, `motion_callbk()`, that responds to left or middle mouse motion — an event triggered when the mouse is moved whilst an appropriate mouse butten is depresses — we would write:

```
XtAddEventHandler(widget, Button1MotionMask | Button2MotionMask,
                  False, motion_callbk, NULL);
```

There are two other steps that need to be done when writing our own event handler. These are:

- Intercepting Events, *and*
- Dispatching Events.

These two steps are basically what the `XtAppMainLoop()` takes care of in normal operation.

Two Xt functions are typically used in this context:

`XtAppNextEvent(XtAppContext, XEvent*)` gets the next event off the event *queue* for a given `XtAppContext` application.

`XtDispatchEvent(XEvent*)` dispatches the event so that callbacks can be invoked.

In between the retrieving of the next event and dispatching this event you may want to write some code that *intercepts* certain events.

Let us look at how the `XtAppMainLoop()` function is coded. Note the comments show where we may place custom application intercept code.

```
void XtAppMainLoop(XtAppContext app)

{ XEvent event;

  for (;;) /* forever */
    { XtAppNextEvent(app, &event);

      /* Xevent read off queue */
      /* inspect structure and intercept perhaps? */

      /* intercept code would go here */

      XtDispatchEvent(&event);
    }
}
```

Chapter 50

The DrawingArea Widget

In this section we will look at how we create and use Motif's DrawingArea Widget which is concerned with the display of graphics. We will also put into practice the Xlib drawing and event scheduling routines met in Chapter 49.

50.1 Creating a DrawingArea Widget

To create a DrawingArea Widget, use `XtVaCreateManagedWidget()` with `xmDrawingAreaWidgetClass` or use `XmCreateDrawingArea()`. Remember to include the `<Xm/DrawingA.h>` header file.

There is also a **DrawnButton** Widget which is a combination of a DrawingArea and a PushButton. There is a `xmDrawnButtonWidgetClass` and definitions are in the `<Xm/DrawnB.h>` header file.

50.2 DrawingArea Resources and Callbacks

A DrawingArea will usually be placed inside a container widget —*e.g.* a Frame or a MainWindow — and it is frequently *scrolled*. As such, it usually inherits size and other resources from its parent widget. You can, however, set resources like `XmNwidth` and `XmNheight` for the DrawingArea directly. The `XmNresizePolicy` resource may also need to be set to allow changes in dimension of the DrawingArea in a program. Possible values are:

`XmRESIZE_ANY` — Allow any size of DrawingArea, `XmRESIZE_GROW` — Allow the DrawingArea to expand only from its initial size,

`XmRESIZE_NONE` — No change in size is allowed.

Three callbacks are associated with this widget:

XmNexposeCallback — This callback occurs when part of the window needs to be redrawn.

XmNresizeCallback — If the dimensions of the window get changed this is called.

XmNinputCallback — If input in the form of a mouse button or key press/release occurs this callback is invoked.

50.3 Using DrawingAreas in Practice

We are now in a position to draw 2D graphics in Motif. Recall that all graphics drawing is performed at the Xlib level and so we have to attach the *higher level* motif widgets to the *lower level* Xlib structures (Chapter 49).

In order to draw anything in a DrawingArea Widget we basically need to do the following:

- Create a DrawingArea widget perhaps contained in other widgets.
- Obtain the Xlib `Display, Window`, *etc.* ID's of the DrawingArea widget (Section 49.1).
- Draw, shade, *etc.* using Xlib graphics routines (Section 49.3).

50.3.1 Basic Drawing — `draw.c`

The `draw.c` program illustrates basic Motif/Xlib drawing principles:

- It creates a DrawingArea, `draw`, contained within a MainWindow.
- We obtain Xlib `Display` and `Screen` IDs with `XtDisplay(draw)` and `XtScreen(draw)`.

- We create a Graphics Context with the `foreground` pixel set to the default `BlackPixel` of the `Screen`.
- We pass the Graphics Context value by making it the **user data** of the `draw` widget. To do this:
  1. Attach the appropriate data to the XmNuserData resource.

     Note: this method can be used to pass any type of data not just graphics as illustrated here.
  2. To retrieve the date use `XtGetValues()`.
  3. The Xlib graphics routines are used to draw lines, rectangles, arcs and text to the DrawingArea (via the `Display` and `Screen` IDs.
- The font for the text labels for each drawing primitive is loaded and set in the function `load_font()`. The `font_height` is computed and used to determine where to place the text labels when drawn by `XDrawString()`.

The full program listing is:

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>

/* Prototype functions */

void quit_call(void);
void draw_cbk(Widget , XtPointer ,
         XmDrawingAreaCallbackStruct *);
void load_font(XFontStruct **);

/* XLIB Data */

Display *display;
Screen *screen_ptr;

main(int argc, char *argv[])
```

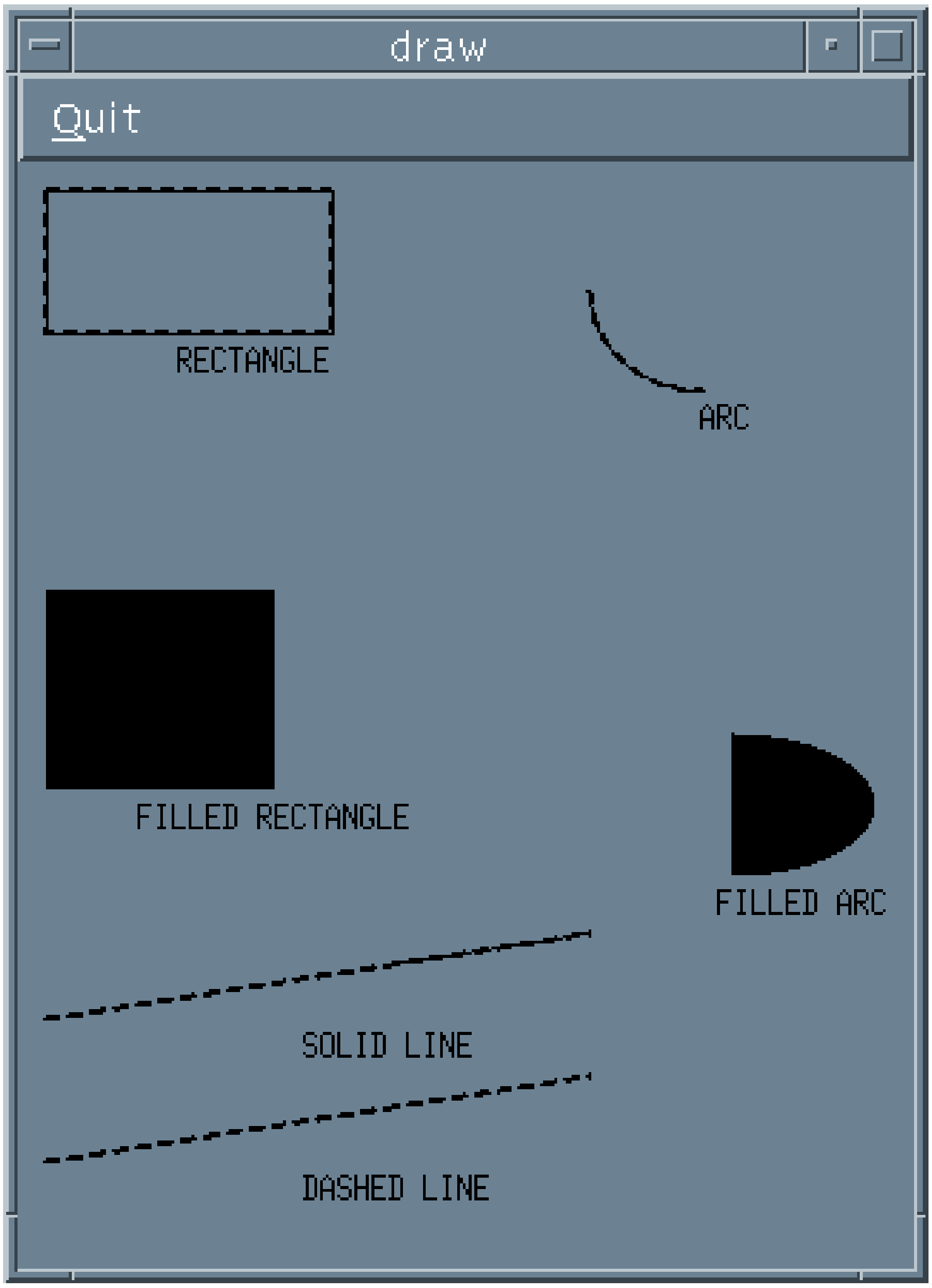


Figure 50.1: Output of `draw.c`

```
{   Widget top_wid, main_w, menu_bar, draw, quit;
    XtAppContext app;
    XGCValues gcv;
    GC gc;


    top_wid = XtVaAppInitialize(&app, "Draw", NULL, 0,
        &argc, argv, NULL,
        XmNwidth,  500,
        XmNheight, 500,
        NULL);

    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass,   top_wid,
        NULL);

    menu_bar = XmCreateMenuBar(main_w, "main_list",
        NULL, 0);
    XtManageChild(menu_bar);

     /* create quit widget + callback */

   quit = XtVaCreateManagedWidget( "Quit",
        xmCascadeButtonWidgetClass, menu_bar,
        XmNmnemonic, 'Q',
        NULL);


    XtAddCallback(quit, XmNactivateCallback, quit_call,
                  NULL);

    /* Create a DrawingArea widget. */
    draw = XtVaCreateWidget("draw",
        xmDrawingAreaWidgetClass, main_w,
        NULL);

    /* get XLib Display Screen and Window ID's  for draw */
    display = XtDisplay(draw);
    screen_ptr = XtScreen(draw);

/* set the DrawingArea as the "work area" of main window */
    XtVaSetValues(main_w,
        XmNmenuBar,    menu_bar,
        XmNworkWindow, draw,
        NULL);

    /* add callback for exposure event */
    XtAddCallback(draw, XmNexposeCallback, draw_cbk, NULL);

    /* Create a GC. Attach GC to the DrawingArea's
       XmNuserData.
       NOTE : This is a useful method to pass data */

    gcv.foreground = BlackPixelOfScreen(screen_ptr);
    gc = XCreateGC(display,
        RootWindowOfScreen(screen_ptr), GCForeground, &gcv);
    XtVaSetValues(draw, XmNuserData, gc, NULL);

    XtManageChild(draw);
    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}


/* CALL BACKS */

void quit_call()

{   printf("Quitting program\n");
    exit(0);
}

/*  DrawingArea Callback. NOTE: cbk->reason says type of
    callback event */
```

```
void
draw_cbk(Widget w, XtPointer data,
         XmDrawingAreaCallbackStruct *cbk)

{   char  str1[25];
    int len1, width1, font_height;
    unsigned int width, height;
    int  x, y, angle1, angle2, x_end, y_end;
    unsigned int line_width = 1;
    int line_style = LineSolid;
    int cap_style = CapRound;
    int join_style = JoinRound;
    XFontStruct *font_info;
    XEvent *event = cbk->event;
    GC gc;
    Window win = XtWindow(w);


  if (cbk->reason != XmCR_EXPOSE)
     { /* Should NEVER HAPPEN for this program */
       printf("X is screwed up!!\n");
       exit(0);
     }

   /* get font info */

   load_font(&font_info);

   font_height = font_info->ascent + font_info->descent;

   /* get gc from Drawing Area user data */

   XtVaGetValues(w, XmNuserData, &gc, NULL);

  /* DRAW A RECTANGLE */

  x = y = 10;
  width = 100;
```

```
height = 50;

XDrawRectangle(display, win, gc, x, y, width, height);

strcpy(str1,"RECTANGLE");
len1 = strlen(str1);
y += height + font_height + 1;
 if ( (x = (x + width/2) - len1/2) < 0 ) x = 10;

XDrawString(display, win, gc, x, y, str1, len1);

/* Draw a filled rectangle */

x = 10; y = 150;
width = 80;
height = 70;

XFillRectangle(display, win, gc, x, y, width, height);

strcpy(str1,"FILLED RECTANGLE");
len1 = strlen(str1);
y += height + font_height + 1;
if ( (x = (x + width/2) - len1/2) < 0 ) x = 10;

XDrawString(display, win, gc, x, y, str1, len1);


/* draw an arc */

x = 200; y = 10;
width = 80;
height = 70;
angle1 = 180 * 64; /* 180 degrees */
angle2 = 90 * 64; /* 90 degrees */

XDrawArc(display, win, gc, x, y, width, height,
         angle1, angle2);
```

```
strcpy(str1,"ARC");
len1 = strlen(str1);
y += height + font_height + 1;
if ( (x = (x + width/2) - len1/2) < 0 ) x = 200;

XDrawString(display, win, gc, x, y, str1, len1);

/* draw a filled arc */

x = 200; y = 200;
width = 100;
height = 50;
angle1 = 270 * 64; /* 270 degrees */
angle2 = 180 * 64; /* 180 degrees */

XFillArc(display, win, gc, x, y, width, height,
         angle1, angle2);

strcpy(str1,"FILLED ARC");
len1 = strlen(str1);
y += height + font_height + 1;
if ( (x = (x + width/2) - len1/2) < 0 ) x = 200;

XDrawString(display, win, gc, x, y, str1, len1);

/* SOLID LINE */

x = 10; y = 300;
/* start and end points of line */
x_end = 200; y_end = y - 30;
XDrawLine(display, win, gc, x, y, x_end, y_end);

strcpy(str1,"SOLID LINE");
len1 = strlen(str1);
y += font_height + 1;
if ( (x = (x + x_end)/2 - len1/2) < 0 ) x = 10;

XDrawString(display, win, gc, x, y, str1, len1);
```

```
  /* DASHED LINE */

  line_style = LineOnOffDash;
  line_width = 2;

  /* set line attributes */

   XSetLineAttributes(display, gc, line_width, line_style,
                     cap_style, join_style);

  x = 10; y = 350;
  /* start and end points of line */
  x_end = 200; y_end = y - 30;
  XDrawLine(display, win, gc, x, y, x_end, y_end);

  strcpy(str1,"DASHED LINE");
  len1 = strlen(str1);
  y += font_height + 1;
  if ( (x = (x + x_end)/2 - len1/2) < 0 ) x = 10;

  XDrawString(display, win, gc, x, y, str1, len1);
 }


void load_font(XFontStruct **font_info)

{  char *fontname = "fixed";
    XFontStruct *XLoadQueryFont();

   /* load and get font info structure  */

  if (( *font_info = XLoadQueryFont(display, fontname))
        == NULL)
      { /* error - quit early */
         printf("%s: Cannot load  %s font\n", "draw.c",
                fontname);
         exit(-1);
    }
}
```

50.3.2 `draw_input1.c` — Input to a DrawingArea

The previous program only illustrated one aspect of the DrawingArea widget, *i.e.* displaying graphics. Another important aspect of this widget is how the widget accepts input. In this Section we will develop a program that illustrate how input is processed in the DrawingArea. We will write a program, `draw_input1.c`, that highlights some deficiencies in the default event handling capabilities within a practical application.

The `draw_input1.c` program accepts mouse input in the DrawingArea. It allows the user to select a colour (as we have seen previously) and then draw a variable size rectangle that is shaded with the chosen colour. A `clear` DrawingArea facility is also provided.

In order to achieve a practical and intuitive manner of user interaction we will need to detect **3** different mouse events (all events described below refer to the *left* mouse button):

- The mouse button **down** indicates that input is about to start and the coordinates selected here are the *top left* corner of the rectangle.
- The mouse button **up** indicates the end of input and a rectangle is drawn using the coordinates selected here — *bottom right* corner.
- It is useful to provide visual feedback as to the size and location of the rectangle as the user is moving the mouse. We can do this by detecting a mouse **motion** and drawing a silhouette of the rectangle as the mouse moves. (See Fig. 50.3). **Note:** This is common practice in many applications where selection of several items, as well a drawing, requires such outlines to be drawn.

To detect mouse clicks *up* and *down* we can use the `XmNinputCallback` resource. However, the default setting of callback resources in a DrawingArea Widget does not allow for mouse motion to be detected as we would like. We therefore have to override the default callback options.

Every Widget has a **Translation Table** (Section 38.8.2) that contains a list of events that it can receive and actions that it is to take upon receipt of

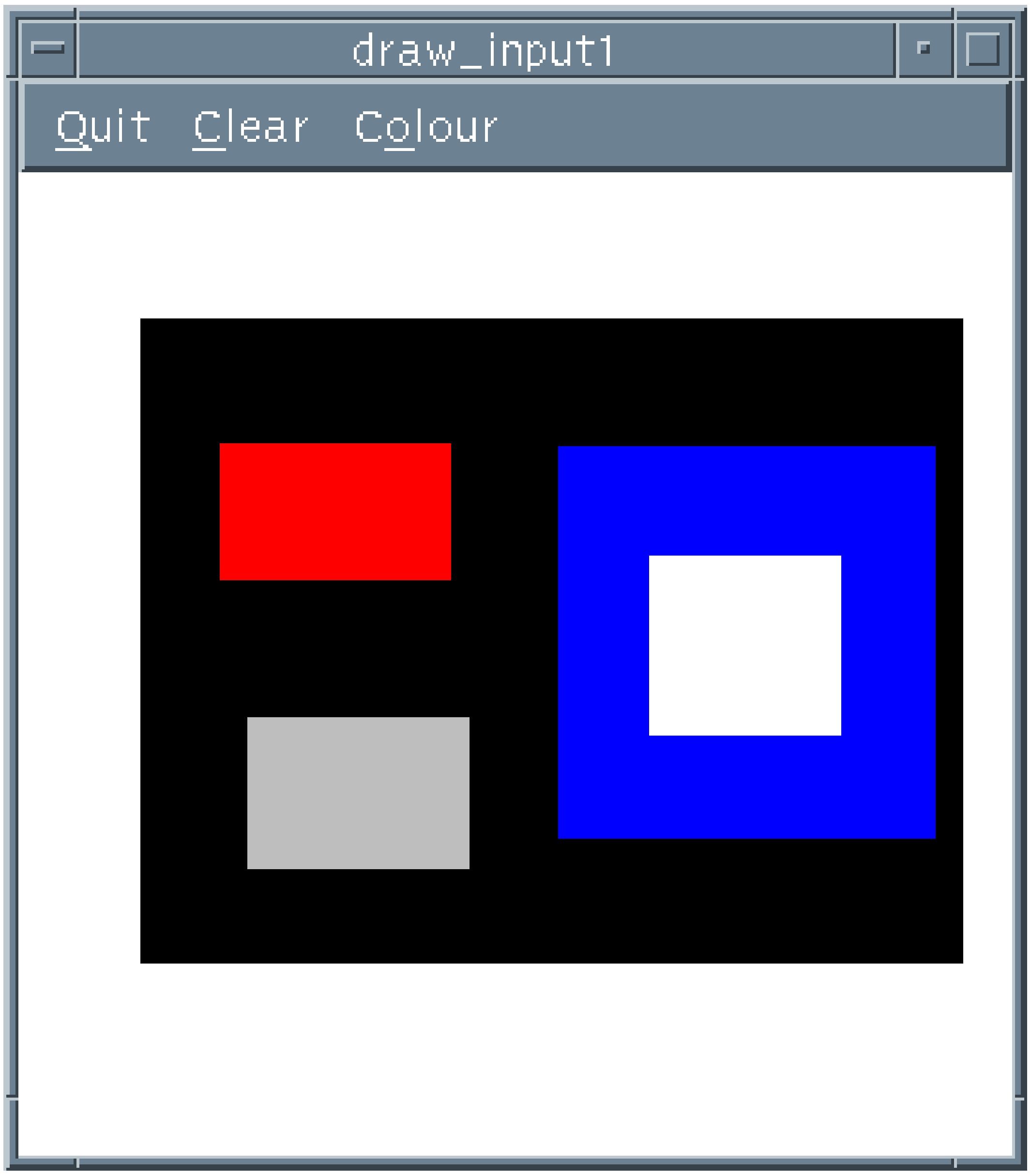


Figure 50.2: Output of `draw_input1.c`

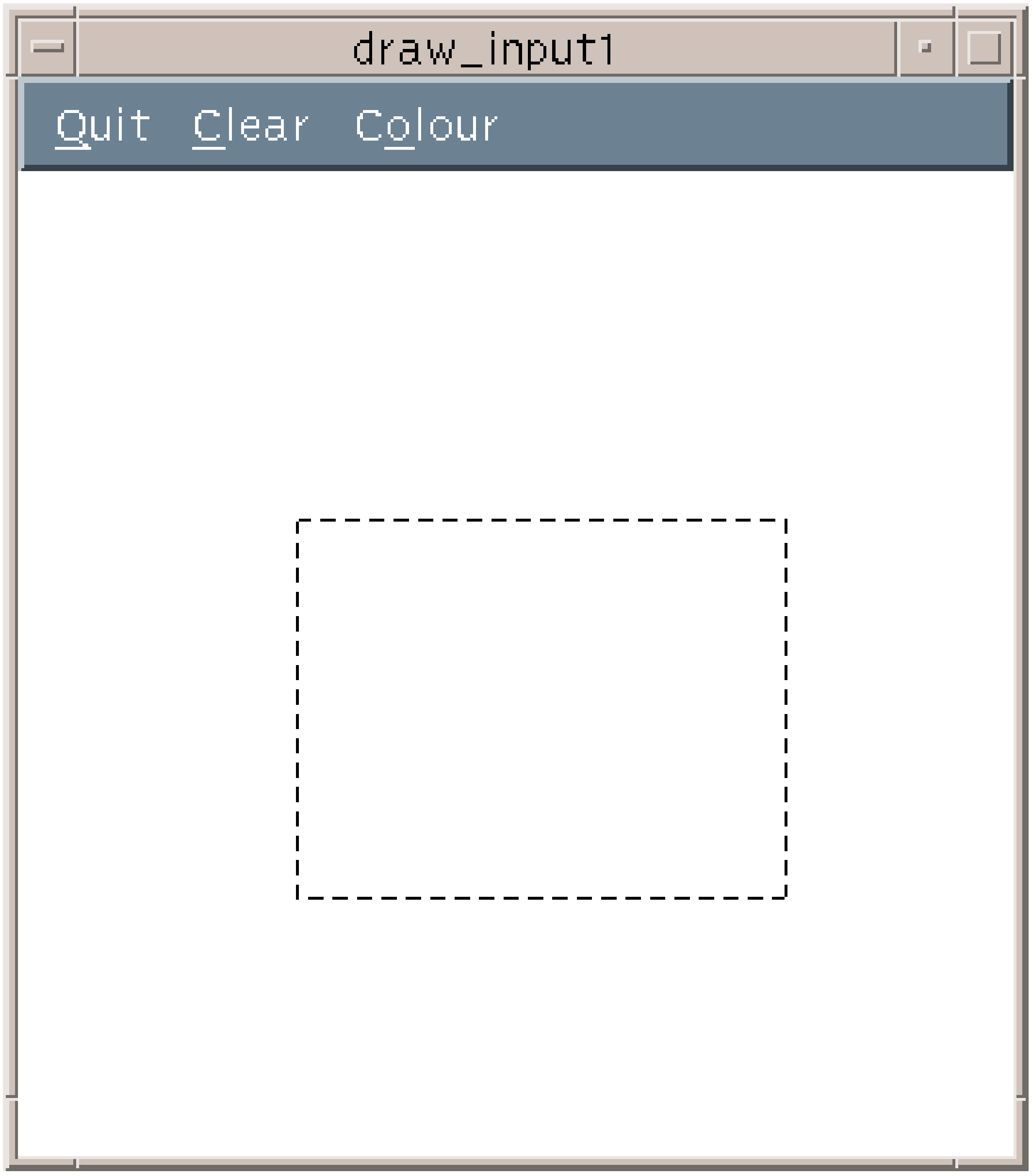


Figure 50.3: Silhouette outline of rectangle during input (`draw_input1.c`)

an event. We basically have to create a new translation table to achieve our desired interaction described above.

We have already defined the translation table format in Section 38.8.2. A translation table consists of events like the below excerpt of the default DrawingArea translation:

```
..............
<Btn1Down>:    DrawingAreaInput() ManagerGadgetArm()
<Btn1Up>:      DrawingAreaInput() ManagerGadgetActivate()
<Btn1Motion>:  ManagerGadgetButtonMotion()
..............
```

Our particular problem is that button motion does not get passed to the `DrawingAreaInput()` function that notifies the program of an input event.

To create a new translation table, for our purpose, we simply include the functions and events we need. In this case:

```
<Btn1Down>:   draw_cbk(down) ManagerGadgetArm()
<Btn1Up>:     draw_cbk(up) ManagerGadgetActivate()
<Btn1Motion>: draw_cbk(motion) ManagerGadgetButtonMotion()
```

where `draw_cbk()` is our callback that performs the drawing. We use the same callback to detect each mouse button down, up or motion action. This is achieved by sending a message to the callback that identifies each action. The arm, activate and motion gadget manager functions control the (default) display of an event action.

In a motif program, we set up our translation table in a `String` structure and use the `XtParseTranslationTable(String*)` to attach a translation table to the `XmNtranslations` resource. We **must** also register the callback with the `actions` associated with the translation events. We use the `XtAppAddActions()` function to do this.

For the above example we create the `String` as follows:

```
String translations =
"<Btn1Motion>: draw_cbk(motion)
               ManagerGadgetButtonMotion() \n\
<Btn1Down>: draw_cbk(down) ManagerGadgetArm() \n\
<Btn1Up>: draw_cbk(up) ManagerGadgetActivate()";
```

and register the callback and create a DrawingArea widget with the correct actions with the following code:

```
actions.string = "draw_cbk";
actions.proc = draw_cbk;
XtAppAddActions(app, &actions, 1);

draw = XtVaCreateWidget("draw",
 xmDrawingAreaWidgetClass, main_w,
 XmNtranslations, XtParseTranslationTable(translations),
 XmNbackground, WhitePixelOfScreen(XtScreen(main_w)),
 NULL);
```

The Callback function would be prototyped by:

`draw_cbk(Widget w, XButtonEvent *event, String *args, int *num_args)`.

On calling the function, we simply inspect the `args[0] String` to see if an `up, down` or `motion` event has occurred and take the approptriate actions described below:

Mouse Down — Simply remember the (x, y) coordinates of the mouse. These are found in the `x,y` elements of the `event` structure.

Mouse Motion — Draw a *dashed* line silhouette of the box whose current size is determined by mouse down (x, y) and current mouse position.

Note: We set the Graphics Context `Logical Function` to `GXinvert` which means that pixels simply get inverted. We must invert them once more to get them back to their original state before we redraw again at another mouse position.

Mouse Up — We finally draw our rectangle with the desired colour.

The complete program listing of `draw_input1.c` is :

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>
```

```
/* Prototype callbacks */

void quit_call(void);
void clear_call(void);
void colour_call(Widget , int);
void draw_cbk(Widget , XButtonEvent *,
               String *, int *);

GC gc;
XGCValues gcv;
Widget draw;
String colours[] = { "Black",  "Red", "Green", "Blue",
                     "Grey", "White"};
long int fill_pixel = 1; /* stores current colour
                         of fill - black default */
Display *display; /* xlib id of display */
Colormap cmap;

main(int argc, char *argv[])

{   Widget top_wid, main_w, menu_bar, quit, clear, colour;
    XtAppContext app;
    XmString  quits, clears, colourss, red, green,
              blue, black, grey, white;
    XtActionsRec actions;


    String translations =
"<Btn1Motion>: draw_cbk(motion)
               ManagerGadgetButtonMotion() \n\
<Btn1Down>: draw_cbk(down) ManagerGadgetArm() \n\
<Btn1Up>: draw_cbk(up) ManagerGadgetActivate()";

    top_wid = XtVaAppInitialize(&app, "Draw", NULL, 0,
        &argc, argv, NULL,
        XmNwidth,  500,
        XmNheight, 500,
        NULL);
    main_w = XtVaCreateManagedWidget("main_window",
        xmMainWindowWidgetClass,   top_wid,
        XmNwidth, 500,
        XmNheight, 500,
        NULL);

    /* Create a simple MenuBar that contains three menus */
    quits = XmStringCreateLocalized("Quit");
    clears = XmStringCreateLocalized("Clear");
    coloursss = XmStringCreateLocalized("Colour");

    menu_bar = XmVaCreateSimpleMenuBar(main_w, "main_list",
        XmVaCASCADEBUTTON, quits, 'Q',
        XmVaCASCADEBUTTON, clears, 'C',
        XmVaCASCADEBUTTON, colourss, 'o',
        NULL);

    XtManageChild(menu_bar);


/* First menu is quit menu -- callback is quit_call() */

    XmVaCreateSimplePulldownMenu(menu_bar, "quit_menu", 0,
        quit_call, XmVaPUSHBUTTON, quits, 'Q', NULL, NULL,
        NULL);
    XmStringFree(quits);

/* Second menu is clear menu -- callback is clear_call() */

    XmVaCreateSimplePulldownMenu(menu_bar, "clear_menu", 1,
        clear_call, XmVaPUSHBUTTON, clears, 'C', NULL, NULL,
        NULL);
    XmStringFree(clears);

     /* create colour pull down menu */

    black = XmStringCreateLocalized(colours[0]);
```

```
red = XmStringCreateLocalized(colours[1]);
green = XmStringCreateLocalized(colours[2]);
blue = XmStringCreateLocalized(colours[3]);
grey = XmStringCreateLocalized(colours[4]);
white = XmStringCreateLocalized(colours[5]);

colour = XmVaCreateSimplePulldownMenu(menu_bar,
    "edit_menu", 2, colour_call,
    XmVaRADIOBUTTON, black, 'k', NULL, NULL,
    XmVaRADIOBUTTON, red, 'R', NULL, NULL,
    XmVaRADIOBUTTON, green, 'G', NULL, NULL,
    XmVaRADIOBUTTON, blue, 'B', NULL, NULL,
    XmVaRADIOBUTTON, grey, 'e', NULL, NULL,
    XmVaRADIOBUTTON, white, 'W', NULL, NULL,
    XmNradioBehavior, True,
    /* RowColumn resources to enforce */
    XmNradioAlwaysOne, True,
    /* radio behavior in Menu */
    NULL);

XmStringFree(black);
XmStringFree(red);
XmStringFree(green);
XmStringFree(blue);
XmStringFree(grey);
XmStringFree(white);


/* Create a DrawingArea widget. */
/* make new actions */

actions.string = "draw_cbk";
actions.proc = draw_cbk;
XtAppAddActions(app, &actions, 1);

draw = XtVaCreateWidget("draw",
 xmDrawingAreaWidgetClass, main_w,
 XmNtranslations, XtParseTranslationTable(translations),
     XmNbackground, WhitePixelOfScreen(XtScreen(main_w)),
     NULL);

    cmap = DefaultColormapOfScreen(XtScreen(draw));
    display = XtDisplay(draw);

/* set the DrawingArea as the "work area" of main window */
    XtVaSetValues(main_w,
        XmNmenuBar,    menu_bar,
        XmNworkWindow, draw,
        NULL);


/* Create a GC. Attach GC to DrawingArea's XmNuserData. */
    gcv.foreground = BlackPixelOfScreen(XtScreen(draw));
    gc = XCreateGC(XtDisplay(draw),
        RootWindowOfScreen(XtScreen(draw)),
        GCForeground, &gcv);

    XtManageChild(draw);
    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

/* CALL BACKS */

void quit_call()

{   printf("Quitting program\n");
    exit(0);
}

void clear_call() /* clear work area */

{ XClearWindow(display, XtWindow(draw));
}
```

```
/* called from any of the "Colour" menu items.
   Change the colour of the label widget.
   Note: we have to use dynamic setting with setargs()..
 */

void
colour_call(Widget w, int item_no)

  /*  w -- menu item that was selected */
  /*  item_no --- the index into the menu */

{
    int n =0;
    Arg args[1];

    XColor xcolour, spare; /* xlib colour struct */

    if (XAllocNamedColor(display, cmap, colours[item_no],
        &xcolour, &spare) == 0)
       return;

    /* remember new colour */
    fill_pixel = xcolour.pixel;
}

/*  DrawingArea Callback.*/

void  draw_cbk(Widget w, XButtonEvent *event,
               String *args, int *num_args)

{   static Position x, y, last_x, last_y;
    Position width, height;

    int line_style;
    unsigned int line_width = 1;
    int cap_style = CapRound;
    int join_style = JoinRound;
```

```
if (strcmp(args[0], "down") == 0)
      {  /* anchor initial point (save its value) */
        x = event->x;
        y = event->y;
      }
 else
   if (strcmp(args[0], "motion") == 0)
      { /* draw "ghost" box to show where it could go */
         /* undraw last box */

         line_style = LineOnOffDash;

         /* set line attributes */

         XSetLineAttributes(event->display, gc,
          line_width, line_style, cap_style, join_style);

         gcv.foreground
           = WhitePixelOfScreen(XtScreen(w));

         XSetForeground(event->display, gc,
                        gcv.foreground);

         XSetFunction(event->display, gc, GXinvert);


        XDrawLine(event->display, event->window, gc,
                   x, y, last_x, y);
        XDrawLine(event->display, event->window, gc,
                   last_x, y, last_x, last_y);
        XDrawLine(event->display, event->window, gc,
                   last_x, last_y, x, last_y);
        XDrawLine(event->display, event->window, gc,
                   x, last_y, x, y);

        /* Draw New Box */
        gcv.foreground
          = BlackPixelOfScreen(XtScreen(w));
        XSetForeground(event->display, gc,
               gcv.foreground);

        XDrawLine(event->display, event->window, gc,
                  x, y, event->x, y);
        XDrawLine(event->display, event->window, gc,
                  event->x, y, event->x, event->y);
        XDrawLine(event->display, event->window, gc,
                  event->x, event->y, x, event->y);
        XDrawLine(event->display, event->window, gc,
                  x, event->y, x, y);
      }

    else
     if (strcmp(args[0], "up") == 0)
      { /* draw full line */

         XSetFunction(event->display, gc, GXcopy);

         line_style = LineSolid;

         /* set line attributes */

         XSetLineAttributes(event->display, gc,
    line_width, line_style, cap_style, join_style);

         XSetForeground(event->display, gc, fill_pixel);

        XDrawLine(event->display, event->window, gc,
                  x, y, event->x, y);
        XDrawLine(event->display, event->window, gc,
                  event->x, y, event->x, event->y);
        XDrawLine(event->display, event->window, gc,
                  event->x, event->y, x, event->y);
        XDrawLine(event->display, event->window, gc,
                  x, event->y, x, y);

        width = event->x - x;
```

```
                height = event->y - y;
                XFillRectangle(event->display, event->window,
                               gc, x, y, width, height);
            }
            last_x = event->x;
            last_y = event->y;

}
```

50.3.3 Drawing to a pixmap — `draw_input2.c`

One problem the `draw_input1.c` program has is that if the window was covered and then exposed the picture would not *redraw* itself (Section 49.6). To see this for yourself run the `draw_input1.c` obscure the MainWindow with another window and then click on the `draw_input1.c` frame to bring it to the foreground and note the appearance of the window.

Indeed, the best method to store the picture we draw is via a `Pixmap`. Since the drawing in this application is an interactive process it would be very difficult to redraw the picture unless we used **Pixmaps**. There is no way that we could predict how the user would use the application and storing each drawing stroke would become complex. The best approach is to store the drawing data in a `Pixmap` by writing directly to a `Pixmap`. Obtaining an immediate visual feedback is also important (the user needs to see what he has drawn) so we also draw directly to the display when data is being input. When an expose event is detected all we need to do is remap the pixmap to the window.

The `draw_input2.c` program does exactly the same task as `draw_input1.c` but draws to a pixmap which can be remapped to the window upon an exposure.

The major differences between the programs are:

- A `Pixmap` is created with the `XCreatePixmap()` function. The `Pixmap` is made the same size as the DrawingArea and is assigned to default `Screen` and `DepthOfScreen` values.
- We still draw some things to the DrawingArea only. The *dashed* boxes which only serve to indicate the current rectangle size **do not** need to be permanently stored and are unlikely to be drawn, covered and re-exposed in between another mouse motion or the mouse up event.

- We draw the filled colour rectangle to **both** DrawingArea and Pixmap so that we get an immediate effect of drawing and we have the Pixmap backup store.
- Upon an expose event we use `XCopyArea()` to copy the Pixmap to the DrawingArea window.

The `draw_input2.c` program listing is as follows:

```
#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>

/* Prototype callbacks */

void quit_call(void);
void clear_call(void);
void colour_call(Widget , int);
void draw_cbk(Widget , XButtonEvent *, String *, int *);
void expose(Widget , XtPointer ,
            XmDrawingAreaCallbackStruct *);

GC gc;
XGCValues gcv;
Widget draw;
Display *display; /* xlib id of display */
Screen *screen;
Colormap cmap;
Pixmap pix;
Dimension width, height; /* store size of pixmap */
String colours[] = { "Black",  "Red", "Green", "Blue",
                     "Grey", "White"};
long int fill_pixel = 1;  /* stores current colour of fill
                             - black default */

main(int argc, char *argv[])

{   Widget top_wid, main_w, menu_bar, quit, clear, colour;
```

```
XtAppContext app;
XmString  quits, clears, colourss, red, green, blue,
          black, grey, white;
XtActionsRec actions;


String translations =
    "<Btn1Motion>: draw_cbk(motion) ManagerGadgetButtonMotion() \n\
     <Btn1Down>: draw_cbk(down) ManagerGadgetArm() \n\
     <Btn1Up>: draw_cbk(up) ManagerGadgetActivate()";

top_wid = XtVaAppInitialize(&app, "Draw", NULL, 0,
    &argc, argv, NULL,
    NULL);

main_w = XtVaCreateManagedWidget("main_window",
    xmMainWindowWidgetClass,   top_wid,
    NULL);

/* Create a simple MenuBar that contains three menus */
quits = XmStringCreateLocalized("Quit");
clears = XmStringCreateLocalized("Clear");
colourss = XmStringCreateLocalized("Colour");

menu_bar = XmVaCreateSimpleMenuBar(main_w, "main_list",
    XmVaCASCADEBUTTON, quits, 'Q',
    XmVaCASCADEBUTTON, clears, 'C',
    XmVaCASCADEBUTTON, colourss, 'o',
    NULL);

XtManageChild(menu_bar);


/* First menu is the quit menu
     -- callback is quit_call() */

XmVaCreateSimplePulldownMenu(menu_bar, "quit_menu", 0,
    quit_call,
    XmVaPUSHBUTTON, quits, 'Q', NULL, NULL,
    NULL);
XmStringFree(quits);

/* Second menu is the clear menu
      -- callback is clear_call() */

XmVaCreateSimplePulldownMenu(menu_bar, "clear_menu", 1,
    clear_call,
    XmVaPUSHBUTTON, clears, 'C', NULL, NULL,
    NULL);
XmStringFree(clears);


/* create colour pull down menu */

black = XmStringCreateLocalized(colours[0]);
red = XmStringCreateLocalized(colours[1]);
green = XmStringCreateLocalized(colours[2]);
blue = XmStringCreateLocalized(colours[3]);
grey = XmStringCreateLocalized(colours[4]);
white = XmStringCreateLocalized(colours[5]);

colour = XmVaCreateSimplePulldownMenu(menu_bar, "edit_menu", 2,
   colour_call,
    XmVaRADIOBUTTON, black, 'k', NULL, NULL,
    XmVaRADIOBUTTON, red, 'R', NULL, NULL,
    XmVaRADIOBUTTON, green, 'G', NULL, NULL,
    XmVaRADIOBUTTON, blue, 'B', NULL, NULL,
    XmVaRADIOBUTTON, grey, 'e', NULL, NULL,
    XmVaRADIOBUTTON, white, 'W', NULL, NULL,
    XmNradioBehavior, True,  /* RowColumn resources set */
    XmNradioAlwaysOne, True, /* radio behavior in Menu */
    NULL);
XmStringFree(black);
XmStringFree(red);
XmStringFree(green);
XmStringFree(blue);
XmStringFree(grey);
XmStringFree(white);


/* Create a DrawingArea widget. */

/* make new actions */

actions.string = "draw_cbk";
actions.proc = draw_cbk;
XtAppAddActions(app, &actions, 1);

draw = XtVaCreateWidget("draw",
    xmDrawingAreaWidgetClass, main_w,
    XmNtranslations, XtParseTranslationTable(translations),
    XmNbackground, WhitePixelOfScreen(XtScreen(main_w)),
    XmNwidth, 500,
    XmNheight, 500,
    NULL);

cmap = DefaultColormapOfScreen(XtScreen(draw));
display = XtDisplay(draw);
screen = XtScreen(draw);

/* Create a GC. Attach GC to the DrawingArea's XmNuserData. */
gcv.foreground = BlackPixelOfScreen(XtScreen(draw));
gc = XCreateGC(XtDisplay(draw),
    RootWindowOfScreen(XtScreen(draw)), GCForeground, &gcv);

/* get pixmap of DrawingArea */
XtVaGetValues(draw, XmNwidth, &width, XmNheight, &height, NULL);

pix = XCreatePixmap(display, RootWindowOfScreen(screen),
                    width, height, DefaultDepthOfScreen(screen));

/* initial white pixmap */
XSetForeground(XtDisplay(draw), gc,
    WhitePixelOfScreen(XtScreen(draw)));
XFillRectangle(display, pix, gc, 0, 0, width, height);

/* reset gc with current colour */
XSetForeground(display, gc, fill_pixel);

/* set the DrawingArea as the "work area" of the main window */
XtVaSetValues(main_w,
    XmNmenuBar,    menu_bar,
    XmNworkWindow, draw,
    NULL);

/* add callback for exposure event */
XtAddCallback(draw, XmNexposeCallback, expose, NULL);

XtManageChild(draw);
XtRealizeWidget(top_wid);
XtAppMainLoop(app);
}

/* CALL BACKS */

void quit_call()

{   printf("Quitting program\n");
exit(0);
}

void clear_call(Widget w, int item_no) /* clear work area */

{ /* clear pixmap with white */
XSetForeground(XtDisplay(draw), gc,
    WhitePixelOfScreen(XtScreen(draw)));

XFillRectangle(display, pix, gc, 0, 0, width, height);

/* reset gc with current colour */
XSetForeground(display, gc, fill_pixel);
    /* copy pixmap to window of drawing area */

    XCopyArea(display, pix, XtWindow(draw), gc,
        0, 0, width, height, 0, 0);
}

/* expose is called whenever all or portions of the drawing area is
   exposed.
 */

void
expose(Widget draw, XtPointer client_data,
       XmDrawingAreaCallbackStruct *cbk)

{   XCopyArea(cbk->event->xexpose.display, pix, cbk->window, gc,
        0, 0, width, height, 0, 0);
}


/* called from any of the "Colour" menu items.
   Change the colour of the label widget.
   Note: we have to use dynamic setting with setargs().
*/

void
colour_call(Widget w, int item_no)

/* w = menu item that was selected
   item_no = the index into the menu
*/

{
    int n =0;
    Arg args[1];

    XColor xcolour, spare; /* xlib color struct */
    if (XAllocNamedColor(display, cmap, colours[item_no],
                         &xcolour, &spare) == 0)
       return;

    /* remember new colour */
    fill_pixel = xcolour.pixel;
}


/*  DrawingArea Callback */


void  draw_cbk(Widget w, XButtonEvent *event,
               String *args, int *num_args)

{
    static Position x, y, last_x, last_y;
    Position width, height;

    int line_style;
    unsigned int line_width = 1;
    int cap_style = CapRound;
    int join_style = JoinRound;



        if (strcmp(args[0], "down") == 0)
          {  /* anchor initial point (i.e., save its value) */
            x = event->x;
            y = event->y;

          }
       else
         if (strcmp(args[0], "motion") == 0)
          { /* draw "ghost" box to show where it could go */

             /* undraw last box */
```

```
     line_style = LineOnOffDash;

     /* set line attributes */

     XSetLineAttributes(event->display, gc, line_width,
                        line_style, cap_style, join_style);

     gcv.foreground = WhitePixelOfScreen(XtScreen(w));
     XSetForeground(event->display, gc, gcv.foreground);

     XSetFunction(event->display, gc, GXinvert);


    XDrawLine(event->display, event->window, gc,
              x, y, last_x, y);
    XDrawLine(event->display, event->window, gc,
              last_x, y, last_x, last_y);
    XDrawLine(event->display, event->window, gc,
              last_x, last_y, x, last_y);
    XDrawLine(event->display, event->window, gc,
              x, last_y, x, y);

    /* Draw New Box */

    gcv.foreground = BlackPixelOfScreen(XtScreen(w));
    XSetForeground(event->display, gc, gcv.foreground);

    XDrawLine(event->display, event->window, gc,
              x, y, event->x, y);
    XDrawLine(event->display, event->window, gc,
              event->x, y, event->x, event->y);
    XDrawLine(event->display, event->window, gc,
              event->x, event->y, x, event->y);
    XDrawLine(event->display, event->window, gc,
              x, event->y, x, y);

  }
```

```
else
 if (strcmp(args[0], "up") == 0)
  { /* draw full line; get GC and use in XDrawLine() */

     XSetFunction(event->display, gc, GXcopy);

     line_style = LineSolid;

     /* set line attributes */

     XSetLineAttributes(event->display, gc,
       line_width, line_style, cap_style, join_style);

     XSetForeground(event->display, gc, fill_pixel);

     XDrawLine(event->display, event->window, gc,
               x, y, event->x, y);
     XDrawLine(event->display, event->window, gc,
               event->x, y, event->x, event->y);
     XDrawLine(event->display, event->window, gc,
               event->x, event->y, x, event->y);
     XDrawLine(event->display, event->window, gc,
               x, event->y, x, y);

    width = event->x - x;
    height = event->y - y;
    XFillRectangle(event->display, event->window, gc,
                   x, y, width, height);

    /* only need to draw final selection to pixmap */

    XDrawLine(event->display, pix, gc,
              x, y, event->x, y);
    XDrawLine(event->display, pix, gc,
              event->x, y, event->x, event->y);
    XDrawLine(event->display, pix, gc,
              event->x, event->y, x, event->y);
    XDrawLine(event->display, pix, gc,
                      x, event->y, x, y);

            XFillRectangle(event->display, pix, gc,
                           x, y, width, height);
          }

          last_x = event->x;
          last_y = event->y;

}
```

50.4 Exercises

NEED SOME EXERCISE — RUN

Chapter 51

Colour

Dealing with colour is one of the most difficult concepts to grasp in X/Motif. In this Chapter we will look at a couple of the basic methods for dealing with colour in Motif. We have already seen some colour examples in action (The `list.c` in Section 45.4 for example).

Readers should note:

- There is a lot more to colour that we briefly address here.
- The main purpose here is to introduce key concepts and illustrate a few sample uses of colour.

51.1 Why is colour so complex?

The X Window system was designed to run on many different computer systems over a network in a device independent manner. However, each system may implement colour in a different way. Some graphics workstations may include specialised graphics hardware, whilst others may have minimal colour support. X tries to deal with this in a generic way, in fact it can (in theory, at least) support devices that range from black/white, monochrome (*grey scale*) through to *full* 24-bit colour. Note that the *quality* of the intended GUI display cannot be guaranteed on minimal configurations, but at least a close resemblance to the GUI will be rendered by X.

51.2 Colour Basics

Colour in X is based on the *RGB Model*: where the colour is determined by *red, green* and *blue* values. This is just like a TV set or computer monitor where an individual dot on the screen(a picture element or *pixel*) has its colour controlled by RGB intensities:

- Absence of all RGB intensities gives **black**.
- Full RGB intensities give **white**.
- **Red** is absence of any blue and green with different *shades* given by the *red* value.
- Shades of **grey** are equal RGB intensities.
- **Purple** shades are mixtures of red/blue and no green *etc.*

51.3 Displaying Colour

We have already remarked that colour displays, available to X, can vary quite a lot although they generally fall into a few classes:

Black/White — where 1 bit per pixel is used. The bits are sometimes called *planes.*

8-bit displays — True colour displays tend to be expensive as they need a lot of memory and involve more complex programming to handle the large amounts of data present. 8-bit displays are the most common. These are capable of only supporting 256 colours at any one time. However, the 256 colours can be chosen from the whole palette of the true colour range. *Colourmaps* or *colour look up tables* are used to enable this (*see* Below).

True Colour or Full Colour — 24 bits per pixel (8 bits each for RGB band) are used to display around 16.5 million possible colours at the same time.

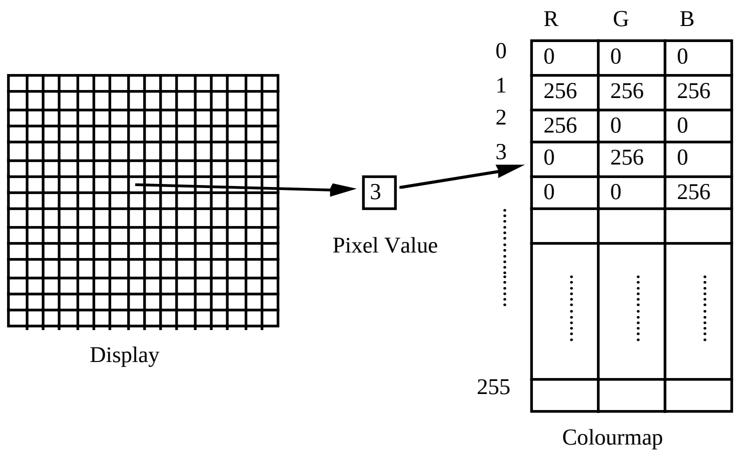


Figure 51.1: Colourmaps and pixels

51.3.1 Colourmaps

A colourmap consists of cells each of which contains 8 bit red, green and blue values. The maximum length of a colourmap is usually 256 entries that correspond to a complete 8-bit colour display.

A pixel value in an image or on the screen basically provides an index to the colourmap which says how to render the appropriate colour. Figure 51.1 shows an example where a pixel value of 3 corresponds to a **green** entry in the colourmap at index 3.

51.4 Colour in X/Motif

In order to be able to support the various means of displaying colour, X allows the setting of colour in a variety of ways. Recall that the Xlib part of X performs all colour operation.

At the heart of all colour operations is the `XColor` structure, which uses 3 basic elements:

`pixel` — the colour index to the colourmap. It is an `unsigned long` data type.

`red, green, blue` — Direct coding of RGB values. The values of theses can range from 0 (off) - 65535 (full) as they are defined as `unsigned shorts`.

`flag` — allow specification of which RGB values are used. `Or` together `DoRed, DoGreen, DoBlue` as required.

There are 2 basic methods that can be used to allocate colour in X.

51.4.1 Colour Database

X provides a database of colours. We simply refer to these by *name* and X will find the appropriate RGB values stored in the database for the given entry. We have in fact been using this style of colour programming in previous examples (Section 45.4).

The colours stored in the database are comprehensive ranging form `red, green, grey, black ....` to more exotic colours like `light salmon` and `tomato`.

The function we typically use to perform this is:

```
XAllocNamedColor(display, cmap, colour_name, &xcolour, &spare);
```

This assigns the appropriate `pixel` value in the `xcolour` (`XColour` data type) structure for a given `colour_name` (`String`) naming a colour stored in a specified `Colourmap`, `cmap`.

We would normally use the default colormap which is obtained for a given widget, `w`, via:

```
cmap = DefaultColormapOfScreen(w);
```

51.4.2 Explicit Colour Coding

If we wish to have greater control over colour, then we may decide to program the RGB values directly. Basically we do this by setting the RGB values of a given `XColour` structure directly, and then set this as an entry in the colourmap.

The function:

```
XAllocColor(Display, Colormap, XColour);
```

will set the `Colormap` RGB entries for a given index from the `pixel, red, green` and `blue` values encoded in the `XColor` structure.

It is good practice to `free` the colormap entry before assigning new values. The following Xlib function is typically used:

```
XFreeColors(Display, Colormap, unsigned long pixel_index,
            int num_pixels, unsigned long planes_to_be_freed);
```

51.4.3 The Colour.c program

Let us look at an example program `colour.c` that illustrates the points raised in the previous sections.

The `colour.c` performs the following tasks:

- It creates a `Label` widget.
  - The `Label` can have its colour changed.
- 3 `Scale` widgets (one for each RGB) change the colour.
- A `Form` widget contains the `Label` and a `RowColumn` widget.
  - The `RowColumn` widget contains the 3 `Scale widgets` (Fig. 51.2).
- Only one colourmap value (index 1) is altered as this is the colourmap index assigned to the `Label`.
- The same callback function is used for the 3 `Scales`. The client data `DoRed` *etc.* being used to distinguish which Scale called the function. The `Scale` callbacks are activated on a `drag` or `ValueChanged` event.
  - The `value` of the callback structure is then used to set the new RGB value.
  - The functions `XAllocColor()` and `XFreeColors()` set up the colormap values for a `DefaultColorMapOfScreen()` colormap, `cmap`.
  - Finally we reset the background colour pixel of the `Label` widget with `XtVaSetValues()`.

The `colour.c` program listing is as follows:

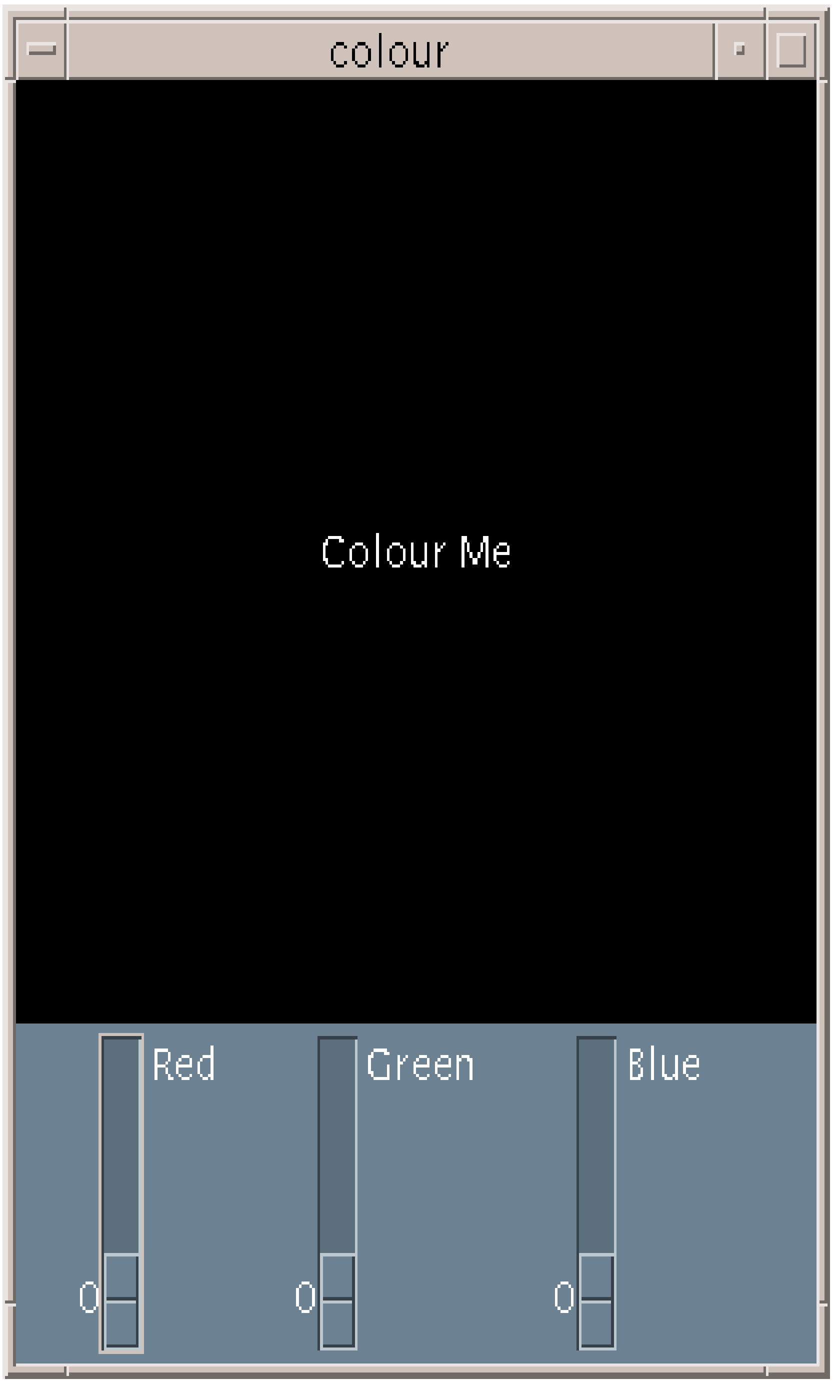


Figure 51.2: Output of `colour.c`

```
#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/RowColumn.h>
#include <Xm/Scale.h>
#include <Xm/Label.h>

/* prototype colour_change function */

void change_colour(Widget , int ,
                   XmScaleCallbackStruct *);

/* Globals */

Widget label; /* this widget gets coloured by
                 slider values for RGB */
XColor color; /* the current colour of the label */

main(int argc, char *argv[])

{
    Widget        top_wid, form, rowcol, scale;
    XtAppContext  app;
    XmString      label_str, red, blue, green;

    top_wid = XtVaAppInitialize(&app, "Colour", NULL, 0,
        &argc, argv, NULL, NULL);

    if (DefaultDepthOfScreen(XtScreen(top_wid)) < 2) {
        puts("You must be using a color screen.");
        exit(1);
    }


    /* Set colour flags field for full RGB display */

    color.flags = DoRed|DoGreen|DoBlue;

    /* initialize first colour */
```

```
XAllocColor(XtDisplay(top_wid),
    DefaultColormapOfScreen(XtScreen(top_wid)), &color);

/* build form to contain label and rowcolumn */

form = XtVaCreateManagedWidget("form",
    xmFormWidgetClass, top_wid,
    NULL);

label_str = XmStringCreateLocalized("Colour Me");

label = XtVaCreateManagedWidget("Label",
    xmLabelWidgetClass,   form,
    XmNlabelString, label_str,
    XmNheight,     300,
    XmNwidth,      300,
    XmNbackground, color.pixel,
    /* Form Attachment resources */
    XmNtopAttachment, XmATTACH_FORM,
    XmNleftAttachment, XmATTACH_FORM,
    XmNrightAttachment, XmATTACH_FORM,
    NULL);

XmStringFree(label_str);

/* Build rowcolumn to contain 3 scales for RGB input */

rowcol = XtVaCreateWidget("rowcol",
    xmRowColumnWidgetClass, form,
    XmNorientation, XmVERTICAL,
    /* Form Attachment resources */
    XmNtopAttachment, XmATTACH_WIDGET,
    XmNtopWidget, label,
    XmNbottomAttachment, XmATTACH_FORM,
    XmNleftAttachment, XmATTACH_FORM,
    XmNrightAttachment, XmATTACH_FORM,
    NULL);
```

```
red = XmStringCreateLocalized("Red");

/* reuses scale widget variable */

scale = XtVaCreateManagedWidget("Red",
    xmScaleWidgetClass, rowcol,
    XmNshowValue, True,
    XmNorientation, XmHORIZONTAL,
    XmNmaximum, 255,
    XmNtitleString, red,
    NULL);

XmStringFree(red);

/* Trap scale valuechange and drags for callbacks */

XtAddCallback(scale, XmNdragCallback,
              change_colour, DoRed);
XtAddCallback(scale, XmNvalueChangedCallback,
              change_colour, DoRed);


green = XmStringCreateLocalized("Green");

scale = XtVaCreateManagedWidget("Green",
    xmScaleWidgetClass, rowcol,
    XmNshowValue, True,
    XmNorientation, XmHORIZONTAL,
    XmNmaximum, 255,
    XmNtitleString, green,
    NULL);

XmStringFree(green);

XtAddCallback(scale, XmNdragCallback,
              change_colour, DoGreen);
XtAddCallback(scale, XmNvalueChangedCallback,
                  change_colour, DoGreen);


    blue = XmStringCreateLocalized("Blue");

    scale = XtVaCreateManagedWidget("Blue",
        xmScaleWidgetClass, rowcol,
        XmNshowValue, True,
        XmNorientation, XmHORIZONTAL,
        XmNmaximum, 255,
        XmNtitleString, blue,
        NULL);

    XmStringFree(blue);

    XtAddCallback(scale, XmNdragCallback,
                  change_colour, DoBlue);
    XtAddCallback(scale, XmNvalueChangedCallback,
                  change_colour, DoBlue);


    XtManageChild(rowcol);

    XtRealizeWidget(top_wid);
    XtAppMainLoop(app);
}

void
change_colour(Widget scale_w, int rgb,
              XmScaleCallbackStruct *cbs)

{
    Colormap cmap = DefaultColormapOfScreen(XtScreen(scale_w));

    /* rgb variable tells us which RGB scale was selected */

    switch (rgb) {
        case DoRed :
            color.red = (cbs->value << 8);
            break;
        case DoGreen :
            color.green = (cbs->value << 8);
            break;
        case DoBlue :
            color.blue = (cbs->value << 8);
    }

    /* reuse the same color index 1 */

    XFreeColors(XtDisplay(scale_w), cmap, &color.pixel, 1, 0);
    if (!XAllocColor(XtDisplay(scale_w), cmap, &color))
        puts("Couldn't XallocColor!"), exit(1);
    XtVaSetValues(label, XmNbackground, color.pixel, NULL);
}
```

Chapter 52

Motif Style

We have mentioned many times in this book that Motif attempts to enforce a standard style in the *look and feel* of its applications through the window manager and default widget actions. One of the goals of Motif is to provide the user with a consistent and easy to use set of applications. Therefore no application should deviate too far from the prescribed Motif norm. In some instances it may be necessary for the programmer to break certain Motif practices in order to achieve a desired form of user interaction. We illustrated this point when we described input to the DrawingArea widget in Chapter 50.

Motif's basic style is based on the (IBM) CUA guidelines. Motif provides many default actions for widgets that adhere to standard GUI practices as layed down by these guidelines. However, when creating widgets the programmer still has some freedom in how the GUI is constructed. For example, menus, menu bars, mouse actions and dialog can take on many different forms. In order to maintain consistency across applications Motif provides a set of guidelines that provide a framework for application development, widget designers, user interface designers and window manager designers. These guidelines are published in the *Motif Style Guide*. This Chapter summarises the important features in the *Motif Style Guide* that are relevant to the application developer.

52.1 The *Motif Style Guide*

As we have just mentioned the *Motif Style Guide* is a document that is aimed at four specific audiences: application development, widget designers, user interface designers and window manager designers. A such the guide is divided up into a number of sections. Not all sections are especially relevant for each audience. Clearly the widget designers and user interface designers will need to be familiar with a majority of the guide, however, application designers need only be familiar with three major aspects of the guide: *User Interface Design Principles*, *Application Design Principles* and *Internationalisation*.

Throughout this text we have already addressed many issues related to the *Motif Style Guide*. We now concentrate on important aspects of style related to specific widgets and general user interaction.

52.2 Menu Style

The *Motif Style Guide* provides many guidelines on the usage of menus in Motif. Menus are usually organised (Chapter 42) in a MenuBar of a MainWindow widget. Motif style *insists* that:

- The MenuBar must be placed horizontally at the top of edge of the application just below the title area of the MainWindow window frame.
- The MenuBar must contain only CascadeButtons with PullDownMenus connected to them. Other classes of button inhibit menu browsing.
- Each menu in the MenuBar should have a single letter *mnemonic* (Section 1.8) attached for easy keyboard selection and each mnemonic should be indicated by underlining.

Many applications have a general functionality, in which case the following general MenuBar menus should be provided with the given mnemonics (in brackets):

File (`Meta-F`) — This menu should contain selections for performing file handling actions such as creating, opening, saving, closing and printing. It should also contain actions for handling the documents as a whole, such as quitting. The following items and mnemonics are common:

New (`Meta-N`) — Create a new file.

Open... (`Meta-O`) — Open a file. A FileSelectionDialog usually appears to allow the user to choose a file.

Save (`Meta-S`) — Save the current file to its current name.

Save As... (`Meta-A`) — Save the current file to a different name

Print... (`Meta-P`) — Print the current file.

Close (`Meta-C`) — Close the window. This option must only be supplied in applications that have *multiple* independent primary windows. The action must only close the current primary window and associated child windows.

Exit (`Meta-x`) — Quit the application.

Selected (`Meta-S`) — This menu should contain selections for objects currently selected by the application. Many items are common with the **File** menu options but are specific to the selected item. Options covered by the **Edit** menu (below) should not be included in this menu. The **Selected** menu is not that commonly used.

Edit (`Meta-E`) — This menu should contain options for performing actions on the current data of the application. This menu uses Motif's *keyboard accelerators* for selection of some items. Keyboard accelerators are much like mnemonics except that they are arbitrary key combinations. The *control* (`Ctrl`), *Shift* and *Alt* keys in combination with other keys are typically used and the particular key assignment model may vary depending on local computer keyboard configurations or host computer compatibility requirements. For example, Apple Macintosh computers have a different cut, paste, copy and undo accelerator model that adheres to standard Macintosh practice for these actions.

In order to set the keyboard accelerator for an appropriate menu item the resources `XmNaccelerator` (`String`) and `XmNacceleratorText` (`XmString`) to set the key combination and menu item label respectively. For example `Ctrl<key>/` is the string used to set the `XmNaccelerator` resource for the `Ctrl-/` key selection combination.

Common actions, with applicable keyboard accelerators or mnemonics denoted in brackets, include:

Undo (`Alt-Backspace`) — Undo the last user operation.

Cut (`Shift-Del`) — Remove the selected portion of data and store it in the *clipboard*. The clipboard is a Motif resource which provides a localised area of memory that allows easy transfer of data within and between applications via *cut and paste* actions.

Copy (`Ctrl-Ins`) — Copy the selected portion of data and store it in the clipboard.

Paste (`Shift-Ins`) — Copy the current contents of the clipboard to the current location.

Clear (`Meta-e`) — Remove the selected data *without* copying it to the clipboard. The remaining data is not compressed to the space left by this operation.

Delete (`Meta-D`) — Remove the selected data *without* copying it to the clipboard, moving following data to fill the space left by this operation.

Select All (`Ctrl-/`) — Select all elements in current work area.

Deselect All (`Ctrl-\`) — Deselect selected items.

View (`Meta-V`) — This menu should contain options allowing the user to change the display of data. The exact makeup of this menu will depend on the application but can include options to control the order of the displayed data, the amount of data displayed or the appearance of the data. For example, in the CDE desktop *file manager* you can list files by icon or by text and files may be listed in alphabetical order or by date.

Options (`Meta-O`) — This menu should contain options allowing the user to customise the application. Once more this menu is very application dependent. One common example is providing facilities that allow the user to change the colours used by the application.

Help (`Meta-H`) — This menu should contain items that provide the user with facilities that guide and inform the user to effectively use the application. DialogBox widgets are usually used to convey the **Help** information. The **Help** menu must be placed on the far right of the MenuBar (Chapter 42).

Motif Style prescribes two acceptable models for the **Help** menu. One is based on providing help on various components of the application such as how to use windows, function mnemonic and accelerator keys. Tutorial information may also be specified. The other model is similar to a user manual approach providing an overview, index and tutorial information.

In addition to prescribing the organisation and formatting of menus the style guide also provides some general guidelines on menu design. Briefly these are:

- Keep menu structures simple.
- Group similar menu elements together.
- List menu items by frequency of use.
- List menu items by order use (which may be more important than frequency ordering).
- Separate destructive actions from frequently chosen items to avoid accidental selection.
- Provide keyboard mnemonics and accelerators for frequently chosen items.
- Provide *tear-off menus* for in frequently used menus (Chapter 42).

52.3 Dialog Widgets

Applications use Dialog widgets to interact with the user. They may simply supply information to the user or may actually canvas for some input (*e.g.* file selection). Applications should only display Dialog boxes when they are required. Motif provides specific Dialog widgets for specific occasions. Examples include ErrorDialog, WarningDialog, CommandDialog, PromptDialog and FileSelectionDialog widgets. The specific usage of Dialog widgets has already been addressed in Chapter 43

52.4 Drag and Drop

Motif provides a variety of guidelines for using its drag and drop facilities. Most of these a provided by default Motif drag and drop actions. Briefly the guidelines are:

- The middle mouse button must be used for drag and drop.
- A drag source can support multiple operations. The user should be able to select the operation that is used. A key selection in combination with the mouse button is typically used:

  `Shift` — selects a move operation.

  `Ctrl` — selects a *copy* operation.

  `Ctrl-Shift` — selects a *link* operation.

  `Esc` — cancels a drag at any time.

  `F1/Help` — should provide *help* information.
- Drag icons should be provided to indicate the three states of the drag operation:
  - A source indicator — to indicate the primary data type for the draggable element.
  - An operation indicator — to indicate whether a move, copy or link operation is occurring . A different icon should represent each type of operation.
  - A state indicator — to indicate whether the a drop is valid.

52.5 Interaction

Motif also provides guidelines on creating applications with consistent interaction. If interaction is more or less uniform across applications then the user can complete basic tasks more quickly. The Motif style guidelines on user interaction are summarised as follows:

Clearly Indicate Actions — Interactions should be made as simple as possible. Providing intuitive visual cues is a good way to achieve this. The following principles should be adopted:

- Use common components — Motif provides a standard appearance for every widget. Every widget also has a standard action. These should never be fundamentally altered by the program. Motif users can then associate a particular component appearance with a particular action or task.
- Use intuitive labels — Most widget can be labelled with a piece of text or graphic. These labels should concisely convey the action of a particular task for the widget it has been assigned to.
- Use graphics to convey action — Graphics can be used to indicate the action of a particular component. For example CascadeButtons use an arrow to point to the direction in which the menu will appear.
- Show default actions — default values should be used for common settings or obvious selections. These values should be set by the application at start up. For example, a RadioButton selection should be set so that the default choice is highlighted when the RadioBox is initially displayed.
- Components should have single modes of operation — the functionality of a component should not change over time in the application.
- Show unavailable components — As the state of the application changes then certain components may become inappropriate. For example certain menu selections. In this case certain selections should be made unavailable by graying the label component (Fig. 52.1). The item should not be removed from the display however. The function `XtSetSensitive(Widget, Boolean)` can be used to set a given `Widget` to be able to receive keyboard or mouse input and to be displayed appropriately. If the `Boolean` value is `True` then the `Widget` can receive input whilst if the value is `False` then no input can be received and the widget will appear *greyed* on the display.

Figure 52.1: A Disabled Menu Element

Provide Feedback — The user should always be informed of the current state of the application. Labels and graphics as described above provide a good means of conveying this information. Typical feedback mechanisms include:

- Showing progress — if an actions takes some time to complete then a WorkingDialog should be displayed. If the application can monitor the progress of the action then it should update the WorkingDialog periodically.
- Providing warnings — Certain actions can cause destructive actions. For example closing an application before saving changes in a current file. The user should be informed of such events via a WarningDialog.
- Providing help — Help facilities should be provided for all aspects interaction no matter how intuitive this may appear.

Allow User Flexibility — The user should be allowed flexibility to adjust elements in the application. Elements that may require adjusting are widget colour, fonts, default values, application parameters, key bindings, labels, messages and even help facilities. Motif resources (Chapter 40) have been especially designed to allow customisation of applications.

52.6 International Markets

As we have already mentioned in Chapter **??** the Internationalisation of Motif has received ever increasing attention. In general many of the issues governing internationalisation are local to a particular system configuration. As we have seen the process of internationalisation can be a difficult process and any tools available to aid this process should be used. Motif provides a broad set of guidelines for the internationalisation of an application. However most of the guidelines will be dependent on the available tools and system configuration. Briefly the Motif style guidelines are as follows:

Text Input — Ideally international text input should be form a keyboard that can provide all the characters of the local language. In some cases a *pre-edit* step is needed where characters are initially typed in an then converted to another set of characters. The system and application programmers should take steps to make sure that this process is well supported and well behaved. Where some conflict or confusion arises over a specific conversion DialogBox should be used to convey the problem and list possible choices in an appropriate menu.

Country Specific Data Formats — Various forms of data need to supported by an application. Examples of data that varies in format across countries include numeric (thousands, millions) and decimal point separators, positive and negative values, currency, date and time formats, telephone numbers, and names and addresses.

Icons, Symbols and Pointers — Graphical symbols cross borders more easily than text labels. However, care may still be needed to make the graphic symbols as cross-cultural as possible.

Scanning Direction — Readers of western languages scan from left to right and from top to bottom. In other languages (*e.g.* Hebrew) readers scan from right to left. This has major implication in the location of components in menus and Dialog widgets. For example, should the help menu be placed to the far right of the MenuBar.

Modularise the Software — If the application software can be partitioned so that the text, input/output modules and other parts of the code are separate then the application should be easier to internationalise as individual modules can be substituted more easily.

Clear Screen Text — Well written screen text is far easier to understand. Screen text should be simple and brief.

Chapter 53

On to C++

In this chapter we provide a stepping stone from C to C++. C++ supports all the features of C, with a few twists and a lot more features thrown in. This chapter starts with a comparison of C and C++, focusing on changes you'll need to make to compile your ANSI C code with an ANSI C++ compiler. It then moves on to some features unique to C++.

Think of C++ as a superset of C. For the most part, every single feature you've come to know and love in C is available in C++ (albeit with a few changes). As in C, C++ programs start with a `main()` function. All of C's keywords and functions work just fine in C++. Even ommand-line arguments `argc` and `argv`, are still supported in C++. In fact, with only a few tweaks here and there, your C programs should run quite well in the C++ world.

53.1 Function Prototypes Are Required

In C, function prototypes are optional, or at least C assumes that they defer to type `int` if they are not declared. As long as there's no type conflict between a function call and the same function's declaration, your program will compile.

In C++, a function prototype *is required* for each of your program's functions. Your C++ program *will not* compile unless each and every function prototype is in place. As in C, you can declare a function without a return type. If no return type is present, the function is assumed to have a return type of `int`.

53.2 Getting C Code to Run under C++

53.2.1 Automatic Type Conversion

C++ uses the same rules as C for automatic type conversion, but with a slight twist.

Although a `void` pointer can be assigned the value of another pointer type without explicit typecasting, the reverse is not true.

For example, although the following code compiles properly in C, it will not compile in C++:

```
void *voidPtr;
short *shortPtr;
voidPtr = shortPtr; /*  This line is just fine... */
shortPtr = voidPtr; /*  This line is fine in C,
                     but WILL NOT compile in C++ */

shortPtr = (short *)voidPtr; /*  This works in C++ */
```

53.2.2 Scope Issues

There are several subtle differences between C and C++ involving scope. A variable's scope defines the availability of the variable throughout the rest of a program. For example, a global variable is available throughout a program, while a local variable is limited to the block in which it is declared. Though C++ follows the same scope rules as C, there are a few subtleties you should be aware of. For example, take a look at the following code. Try to guess the value of size at the bottom of `main()`:

```
char dummy[32];

int main()
{
  long size;
  struct dummy
    {
      char myArray[ 64 ];
```

```
     };

  size = sizeof( dummy );
  return 0;
}
```

In C, `size` ends up with a value of 32; the reference to dummy in the `sizeof()` statement matches the global variable declared at the top of the program.

In C++, however, `size` ends up with a value of 64; the reference to `dummy` matches the `struct` tag inside `main()`.

In C++, a structure name declared in an inner scope can *hide* a name in an outer scope. This same rule holds true for an enumeration:

`enum colour  red, green, blue ;`

In C++, this `enum` creates a type named `colour` that can be used to declare other `enum`s and would obscure a global with the same name.

Here's another example:

```
int main()
{
   struct s
    {
     enum { good, bad, ugly } clint;
    };
   short good;
   return 0;
}
```

An ANSI C compiler will not compile this code, complaining that the identifier good was declared twice. The problem here is with the scope of the enumeration constant good. In C, an enumeration constant is granted the same scope as a local variable, even if it is embedded in a struct definition. When the compiler hits the short declaration, it complains that it already has a good identi-fier declared at that level.

In C++, this code compiles cleanly. Why? C++ enumeration constants embedded in a struct definition have the same scope as that struct's fields. Thus, the enumeration constant good is hidden from the short declaration at the bottom of `main()`. A third example involves multiple declarations of the same variable within the same scope. Consider the following code:

```
short gMyGlobal;
short gMyGlobal; /* Cool in C, error in C++ */
```

The C compiler will resolve these two variable declarations to a single declaration. The C++ compiler, on the other hand, will report an error if it hits two variable declarations with the same name.

It's useful to be aware of the difference between a declaration and a definition. A declaration specifies the types of all elements of an identifier. For example, a function prototype is a declaration. Here are some more declarations:

```
char name[ 20 ];
typedef int myType;
const short kMaxNameLength = 20;
extern char aLetter;
short MyFunc( short myParam );
```

As you can see, a declaration can do more than tie a type to an identifier. A declaration can also be a defini-tion. A definition instantiates an identifier, allocating the appropriate amount of memory. In this declaration:

```
const short kMaxNameLength = 20;
```

the constant `kMaxNameLength` is also defined and initialized.

53.3 New Features of C++

The remainder of this chapter will take you beyond C into the heart of C++. While we won't explore object programming in this chapter, we will cover just about every other C++ concept.

53.3.1 Comment block markers

C's comment block markers, `/*` and `*/`, perform the same func-tion in C++. In addition, C++ supports a single-line comment marker. When a C++ compiler encounters the characters `//`, it ignores the remainder of that line of code. Here's an example:

```
int main()
{
short numGuppies;  // May increase suddenly!!
return 0;
}
```

As you'd expect, the characters `//` are ignored inside a comment block. In the following example, `//` is included as part of the comment block:

```
int main()
{
  /* Just a comment... // */
return 0;
}
```

Conversely, the comment characters `/*` and `*/` have no special meaning inside a single-line comment. The start of the comment block in the following example is swallowed up by the single-line comment:

```
int main()
{
// Don't start a /* comment block
inside a single-line comment...
This code WILL NOT compile!!! */

return 0;
}
```

The compiler will definitely complain about this example!

53.3.2 The `iostream`

In a standard C program, input and output are usually handled by Standard Library routines such as `scanf()` and `printf()`. While you can call `scanf()` and `printf()` from within your C++ program, there is an elegant alternative. The `iostream` facility allows you to send a sequence of variables and constants to an output stream, just as `printf()` does. Also, iostream makes it easy to convert data from an input stream into a sequence of variables, just as scanf() does.

Though the iostream features presented in this section may seem simplistic, don't be fooled. iostream is actually quite sophisticated. In fact, iostream is far more powerful than C's standard I/O facility. The material given here will allow you to perform the input and output you'll need to get through the next few chapters. Later, we'll explore iostream in more depth.

The iostream predefines three streams for input and output. cin is used for input, cout for normal output, and cerr for error output. The << operator is used to send data to a stream. The >> operator is used to retrieve data from a stream. The << operator is known as the insertion operator because it allows you to insert data into a stream. The >> operator is known as the extraction operator be-cause it allows you to extract data from a stream.

Here's an example of the << operator:

```
#include <iostream.h>
int main()
{
  cout << "Hello, world!";
  return 0;
}
```

This program sends the text string Hello, world! to the console, just as if you'd used printf().

The include file <iostream.h> contains all of the definitions needed to use iostream. Since << is a binary operator, it requires two operands. In this case, the operands are cout and the string Hello, world! .

The destination stream always appears on the left side of the << operator. Just like the & and * operators, >> and << have more than one meaning (>> and << are also used as the right and left shift operators). Don't worry about confusion, however. The C++ compiler uses the operator's context to determine which meaning is appropriate.

As with any other operator, you can use more than one << on a single line. Here's another example:

```
#include <iostream.h>
int main()
```

```
{
short i = 20;
cout << "The value of i is " << i;
return 0;
}
```

This program produces the following output:

```
The value of i is 20
```

`iostream` knows all about C++'s built-in data types. This means that text strings are printed as text strings, shorts as shorts, and floats as floats, complete with decimal point. No special formatting is necessary.

An `iostream` Output Example

```
#include <iostream.h>
int main()
{
char *name = "Dr. Marshall";
cout << "char: " << name[ 0 ] << '\n'
<< "short: " << (short)(name[ 0 ]) << '\n'
<< "string: " << name << '\n'
<< "address: " << (unsigned long)name;
return 0;
}
```

The cout Source Code The program starts by initializing the char pointer name, point-ing it to the text string "Dr. Marshall". Next comes one giant statement featuring eleven different occurrences of the << opera-tor. This statement produces four lines of output.

The following line of code

```
cout << "char: " << name[ 0 ] << '\n'
```

produces this line of output:

```
char: D
```

As you'd expect, printing `name[ 0 ]` produces the first character in name, an uppercase D.

The next line of code is

```
<< "short: " << (short)(name[ 0 ]) << '\n'
```

The output associated with this line of code is as follows:

```
short: 68
```

This result was achieved by casting the character 'D' to a short. In general, `iostream` displays integral types (such as short and int) as an integer. As you'd expect, a float is displayed in floating-point format.

The next line of code

```
<< "string: " << name << '\n'
```

produces this line of output:

```
string: Dr. Marshall
```

When the << operator encounters a `char` pointer, it assumes you want to print a zero-terminated string.

The final chunk of code in our example shows another way to display the contents of a pointer:

```
<< "address: " << (unsigned long)name;
```

Again, name is printed, but this time is cast as an unsigned long. Here's the result:

```
address: 2150000
```

As you can see, `cout` does what it thinks makes sense for each type it prints. Later in Chapter 57, we will see how to customize cout by using it to print data in a specified format or teaching it how to print your own data types.

Let us now consider an `iostream` input example

```
#include <iostream.h>
const short kMaxNameLength = 40;
int main()
{
   char name[ kMaxNameLength ];
   short myShort;
   long myLong;
   float myFloat;

   cout << "Type in your first name: ";
   cin >> name;

   cout << "Short, long, float: ";
   cin >> myShort >> myLong >> myFloat;

   cout << "\nYour name is: " << name;
   cout << "\nmyShort: " << myShort;
   cout << "\nmyLong: " << myLong;
   cout << "\nmyFloat: " << myFloat;
   return 0;
}
```

As is always the case when you use `iostream`, the program starts by including the file <`iostream.h`>. Next, the constant `kMaxNameLength` is defined, providing a length for the char array name.

When a variable is defined using the const qualifier, an initial value must be provided in the definition, and that value cannot be changed for the duration of the program. Although some C programmers tend to use `#define` instead of const, C++ programmers prefer const to `#define`.

`cin` uses `cout` and << to prompt for a text string, a short, a long, and a float. `cin` and >> are used to read the values into the four variables `name, myShort, myLong`, and `myFloat`.

The next line uses ¡¡ to send a text string to the console:

```
cout << "Type in your first name: ";
```

Next, ¿¿ is used to read in a text string:

```
cin >> name;
```

When the ¿¿ operator reads a text string, it reads a character at a time until a white space character (like a space or a tab) is encountered.

Next, three more pieces of data are read using a *single* statement. First, display the prompt:

```
cout << "Short, long, float: ";
```

Then, read in the data, separating the three receiving variables by consecutive >> operators:

```
cin >> myShort >> myLong >> myFloat;
```

Be sure to separate each of the three numbers by a space (or some white space character). Also, make sure the numbers match the type of the corresponding variable. For example, it's probably not a good idea to enter 3.52 or 125000 as a short, although an integer like 47 works fine as a float.

Finally, display each of the variables we worked so hard to fill with `cout` calls.

53.3.3 `iostream` and Objects

So far, `iostream` might seem primitive compared to the routines in C's Standard Library. After all, routines like `scanf()` and `printf()` give you precise control over your input and output. Routines like `getchar()` and `putchar()` allow you to process one character at a time, letting you decide how to handle white space.

The `iostream` is very powerful. However, to unleash `iostream`'s true power, you must first come up to speed on object oriented programming. The `iostream` concepts presented here are the bare minimum you'll need to get through the sample programs in the next few chapters. For now, basic input and output are all we need.

53.3.4 Default Argument Initializers

C++ allows you to assign default values (known as *default argument initializers*) to a function's arguments. For example, here's a simple default routine:

```
void Deffun( short default = 40 )
{
// A default of 440 is  assigned if no other value passed in
}
```

If you call this function with a parameter, the value you pass in is used. For example, the call Deffun(30);

will assign a value of 30 to `default`,

If you call the function without specifying a value, the default value is used.

The call `Deffun();`

will assign a value of 40 to `default`,

This technique works with multiple parameters as well, although the rules get a bit more complicated. You can specify a default value for a parameter only if you also specify a default for all the parameters that follow it. For example, this declaration is cool:

```
void GotSomeDefaults( short manny, short moe=2,
char jack='x' );
```

Since the second parameter has a default, the third parameter must have a default.

The next declaration won't compile, however, because the first parameter specifies a default and the parameter that follows does not:

```
void WillNotCompile( long time=100L, short stack );
```

Default parameter values are specified in the function prototype rather than in the function's implementation. For example, here's a function prototype, followed by the function itself:

```
void MyFunc( short param1 = 27 );

void MyFunc( short param1 )
{
// Body of the function...
}
```

53.3.5 Reference Variables

In C, all parameters are passed by value as opposed to being passed by reference. When you pass a parameter to a C function, the value of the parameter is passed on to the function. Any changes you make to this value are not carried back to the calling function.

Here's an example:

```
void DoubleMyValue( short valueParam )
  {
     valueParam *= 2;
   }

int main()
  {
     short number = 10;
     DoubleMyValue( number );
     return 0;
  }
```

`main()` sets `number` to 10, then passes it to the function `DoubleMyValue()`. Since number is passed by value, the call to `DoubleMyValue()` has no effect on number. When `DoubleMyValue()` returns, `number` still has a value of 10.

Here's an updated version of the program:

```
void DoubleMyValue( short *numberPtr )
   {
     *numberPtr *= 2;
    }

int main()
   {
     short number = 10;
     DoubleMyValue( &number );
     return 0;
}
```

In this version, number's `address` is passed to `DoubleMyValue()`. By dereferencing this *pointer*,`DoubleMyValue()` can reach out and change the value of number. When `DoubleMyValue()` returns, number will have a value of 20.

In C++ Reference variables, howvere, allow you to pass a parameter by reference, without using pointers. Here's another version of the program, this time implemented with a reference variable:

```
void DoubleMyValue( short &referenceParam )
   {
     referenceParam *= 2;
   }


 int main()
  {
     short number = 10;
     DoubleMyValue( number );
     return 0;
  }
```

Notice that this code looks just like the first version, with one small exception: `DoubleMyValue()`'s parameter is defined using the `&` operator:

```
short &referenceParam
\begin{verbatim}

The {\tt \&} marks {\tt referenceParam} as a reference variable and tells
the compiler that {\tt referenceParam} and its corresponding input
parameter, number, are one and the same. Since both names refer
to the same location in memory, changing the value of
{\tt referenceParam} is exactly the same as changing number.


Here is an example call by reference program:

\begin{verbatim}
#include <iostream.h>

void CallByValue( short valueParam );
void CallByReference( short &refParam );

int main()
{
short number = 12;
long longNumber = 12L;
cout << "&number:     " <<
(unsigned long)&number << "\n";

cout << "&longNumber: " <<
(unsigned long)&longNumber << "\n\n";

CallByValue( number );
cout << "After ByValue: " << number << "\n\n";

CallByReference( number );
cout << "After ByRef( short ): " << number << "\n\n";

CallByReference( longNumber );
cout << "After ByRef( long ): " << longNumber << "\n";

return 0;
}

void CallByValue( short valueParam )
{
cout << "&valueParam: " <<
 (unsigned long)&valueParam << "\n";
valueParam *= 2;
}

void CallByReference( short &refParam )
{
cout << "&refParam:   " <<
 (unsigned long)&refParam << "\n";
refParam *= 2;
}
```

Reference variables are frequently used as *call-by-reference* parameters. However, they can also be used to establish a link between two variables in the same scope. Here's an example:

```
short romulus;
short &remus = romulus;
```

The first line of code defines a `short` with the name `romulus`. The second line of code declares a reference variable with the name `remus`, linking it to the variable `romulus`. Just as before, the `&` marks remus as a reference variable.

Now that `remus` and `romulus` are linked, they share the same location in memory. Changing the value of one is exactly the same as changing the value of the other.

It's important to note that a reference variable **must** be initialized with a variable as soon as it is declared.

The following code will not compile:

```
short romulus;
short &remus; // Will not compile!!!
remus = romulus;
```

The reference variable **must** also be of the same type as the variable it references. The following code won't work:

```
short romulus;
long &remus = romulus; // Type mismatch!!!
```

In addition, once established, the link between a reference and a regular variable cannot be changed as long as the reference remains in scope. In other words, once `remus` is linked to `romulus`, it *cannot* be set to reference a different variable.

53.3.6 Function Name Overloading

Function name overloading, allows you to write several functions that share the same name. Suppose you needed a function that would print the value of one of your variables, be it long, short, or a text string. You could write one function that takes four parameters:

```
Display( short whichType,
         long longParam,
         short shortParam,
         char *textParam );
```

The first parameter might act like a switch, determining which of the three types you were passing in for printing. The main code of the function might look like this:

```
if ( whichType == kIsLong )
   cout << "The long is: " << longParam << "\n";
 else if ( whichType == kIsShort )
     cout << "The short is: " << shortParam << "\n";
    else if ( whichType == kIsText )
       cout << "The text is: " << text << "\n";
```

Another solution is to write three separate functions, one for printing shorts, one for longs, and one for text strings:

```
void DisplayLong( long longParam );
void DisplayShort( short shortParam );
void DisplayText( char *text );
```

Each of these solutions has an advantage. The first solution groups all printing under a single umbrella, making the code somewhat easier to maintain. On the other hand, the second solution is more modular than the first. If you want to change the method you use to display longs, you modify only the routine that works with longs; you don't have to deal with the logic that displays other types.

As you might expect, there is a third solution that combines the benefits of the first two. Here's how it works.

As mentioned earlier, C++ allows several functions to share the same name by way of function name overloading. When an overloaded function is called, the compiler compares the parameters in the call with the parameter lists in each of the candidate functions. The candidate with the most closely matching parameter list is the one that gets called.

A function's parameter list is also known as its signature. A function's name and signature combine to distinguish it from all other functions. Note that a function's return type is not part of its signature.

Here's the third solution that takes advantage of function name overloading:

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

```
void Display( short shortParam );
void Display( long longParam );
void Display( char *text );

int main()
{
short myShort = 3;
long myLong = 12345678L;
char *text = "Make it so...";

Display( myShort );
Display( myLong );
Display( text );

return 0;
}

void Display( short shortParam )
{
cout << "The short is: " << shortParam << "\n";
}

void Display( long longParam )
{
cout << "The long is:  " << longParam << "\n";
}

void Display( char *text )
{
cout << "The text is:  " << text << "\n";
}
```

The output of this program is:

```
The short is: 3
The long is: 12345678
The text is: Make it so...
```

The program starts with three function prototypes, each of which shares the name `Display()` Notice that each version of Display() has a unique signature. This is important. You are **not allowed** to define two functions with the same name and the same signature.

main() starts by defining three variables: `myShort,` `myLong` and a `text` string.

Next, `Display()` is called three times.

- First, a `myShort` is passed as a parameter. Since this call exactly matches one of the `Display()` routines, the compiler doesn't have a problem deciding which function to call.
- Similarly, the calls passing `myLong` and a `text` string to `Display()` match perfectly with the `Display()`functions having `long` and `text` string signatures, repectively.

Matching Rules for Overloaded Functions

The above example was fairly straightforward. The compiler had no difficulty deciding which version of `Display()` to call because each of the calls matched perfectly with one of the `Display()` functions.

But what do you think would happen if you passed a float to `Display()`?

```
Display( 1.0 );
```

When the compiler can't find an exact match for an over-loaded function call, it turns to a set of rules that determine the best match for this call. After applying each of the rules, unless one and only one match is found, the compiler reports an error.

As you've already seen, the compiler starts the matching process by looking for an exact match between the name and signature of the function call and a declared function. If a match is not found, the compiler starts promoting the type of any integral parameters in the function call, following the rules for automatic type conversion similar to C. For example, a `char` or a `short` would be promoted to an `int` and a `float` would be promoted to a double.

If a match is still not found, the compiler starts promoting non-integral types. Finally, the ellipsis operator in a called function is taken into account, matching against zero or more parameters. In answer to our earlier question, passing a float to `Display()` would result in an error, listing the function call as ambiguous. If we had written a version of `Display()` with a `float` or a `double` in its signature, the compiler would find the match.

53.3.7 The `new` and `delete` Operators

In C, memory allocation typically involves a call to `malloc()` paired with a call to free() when the memory is no longer needed. In C++, the same functionality is provided by the operators `new` and `delete`.

Call `new` when you want to allocate a block of memory.

For example, the following code allocates a block of `1024 chars`:

```
char *buffer;
buffer = new char[1024];
```

`new` takes a type as an operand, allocates a block of memory the same size as the type, and returns a pointer to the block.

To return the memory to the heap, use the `delete` operator. The next code frees up the memory just allocated:

```
delete [] buffer;
```

The brackets (`[]`) in the preceding line of code indicate that the item to be deleted is a pointer to an array. If you are deleting something other than a pointer to an array, leave the brackets out, for example:

```
int myIntPtr;
myIntPtr = new int;
delete myIntPtr;
```

`new` can be used with any legal C++ type, including those you create yourself.

Every program that allocates memory runs the risk that its request for memory will fail, most likely because there's no more memory left to allocate. If your program uses new to allocate memory, it had better detect, and handle, any failure on new's part.

Since `new` returns a value of 0 when it fails, the simplest approach just checks this return value, taking the appropriate action when new fails:

```
char *bufPtr;
bufPtr = new char[ 1024 ];

if ( bufPtr == 0 )
   cout << "Not enough memory!!!";
else
   DoSomething( bufPtr );
```

This code uses `new` to allocate a 1024-byte buffer. If new fails, an error message is printed; otherwise, the program goes on its merry way.

This approach requires that you check the return value every time you call new. If your program performs a lot of memory allocation, this memory-checking code can really add up. As your programs get larger and more sophisticated, you might want to consider a second strategy.

C++ allows you to specify a single routine, known as a new handler, that gets called if and when `new` fails. Design your new handler for the general case so that it can respond to any failed attempt to allocate memory.

Whether or not you designate a new handler, `new` will still return 0 if it fails. This means you can design a two-tiered memory management strategy combining a new handler and code that runs if `new` returns 0.

To specify a new handler, pass the handler's name to the function `set_new_handler()`. To use `set_new_handler()`, be sure to include the file <new.h>:

```
#include <new.h>

void NewFailed( void );

int main()
{
set_new_handler( NewFailed );
.
.
.
}
```

One possible memory allocation strategy is to allocate a block of memory at the beginning of your program, storing a pointer to the block in a global variable. Then, when new fails, your program can free up the spare memory block, ensuring that it will have enough memory to perform any housekeeping chores that it requires in a memory emergency. A new Example Our next sample program repeatedly calls new until the program runs out of memory, keeps track of the number of memory re-quests, and then reports on the amount of memory allocated before failure. This program uses the spare memory scheme just described.

The code listing is as follows:

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

void NewFailed();

char gDone = false;
char *gSpareBlockPtr = 0;

int main()
{
char *myPtr;
long numBlocks = 0;

cout << "Installing NewHandler...\n";

set_new_handler( NewFailed );
gSpareBlockPtr = new char[20480];

while ( gDone == false )
{
myPtr = new char[1024];
numBlocks++;
}

cout << "Number of blocks allocated: " << numBlocks;

return 0;
}

void NewFailed()
```

```
{
if ( gSpareBlockPtr != 0 )
{
delete gSpareBlockPtr;
gSpareBlockPtr = 0;
}

gDone = true;
}
```

The number of blocks you can allocate before you run out of memory depends on the amount of memory you make available to your program.

`NewFailed()` is the function we want called if new fails in its attempt to allocate memory:

53.3.8 The Scope Resolution Operator, `::`

C++'s scope resolution operator is denoted by `::`. The scope resolution operator precedes a variable, telling the compiler to look outside the current block for a variable of the same name.

Suppose you declare a global variable and a local variable with the same name:

```
short number;

int main()
{
   short number;
   number = 5; // local reference
   ::number = 10; // global reference
   return 0;
}
```

Inside `main()`, the first assignment statement refers to the local definition of `number`. The second assignment statement uses the scope operator to refer to the global definition of `number`. This code leaves the *local* `number` with a value of 5 and the *global* `number` with a value of 10.

Many programmers start all their global variables with a lowercase g to differentiate them from local variables. If you use this convention, you'll never be in the situation where a local variable is obscuring a global variable. This doesn't mean you should ignore the scope resolution operator As we get into object programming, we'll find that the scope resolution operator is invaluable.

A Scope Resolution Operator Example

Our next sample program offers a quick demonstration of the scope resolution operator.

```
#include <iostream.h>

short myValue = 5;

int main()
{
short yourValue = myValue;

cout << "yourValue: " << yourValue << "\n";

short myValue = 10;
yourValue = myValue;

cout << "yourValue: " << yourValue << "\n";

yourValue = ::myValue;
cout << "yourValue: " << yourValue << "\n";

return 0;
}
```

The output of this program is:

```
yourValue: 5
yourValue: 10
yourValue: 5
```

First we define a global variable with the name `myValue`, initializing it to a value of 5: The we define a local variable named `yourValue` and assigns it the value in `myValue`.

Then, yourValue is printed out, showing it with a value of 5, the same as the global myValue:

Next, a local with the name `myValue` is defined and initialized with a value of 10. When `myValue` is copied to `yourValue` which variable is copied, the local or the global?

As you can see from the output, the reference to `myValue` matches with the local declaration, showing yourValue with a value of 10:

Then, the scope resolution operator is used to copy `myValue` to yourValue. When yourValue is printed again, it has a value of 5, showing that `::myValue` refers to the global declaration of

The scope resolution operator can be applied only when a match is available. Applying the scope resolution operator to a local variable without a corresponding global will generate a compile error.

53.3.9 Inline Functions

Traditionally, when a function is called, the CPU executes a set of instructions that move control from the calling function to the called function. Tiny as these instructions may be, they still take time. C++, however, provides *inline functions*, which allow you to bypass these instructions and save a bit of execution time.

When you declare a function using the *inline* keyword, the compiler copies the body of the function into the calling function, making the copied instructions a part of the calling function as if it were written that way originally. The benefit to you is a slight improvement in performance. The cost is in memory usage:

- If you call an inline function twenty times from within your program, twenty copies of the function will be grafted into your object code.

An Inline Function Example

In order to understabd inline function consider the following example program. The program features a single inline function that returns the value achieved when its first argument is raised to its second argument's power.

The listing for `inline.cpp` is as follows:

```
#include <iostream.h>

inline long power( short base, short exponent );

int main()
{
cout << "power( 2, 3 ): " <<
power( 2, 3 ) << "\n";

cout << "power( 3, 6 ): " <<
power( 3, 6 ) << "\n";

cout << "power( 5, 0 ): " <<
power( 2, 0 ) << "\n";

cout << "power( -3, 4 ): " <<
power( -3, 4 ) << "\n";

return 0;
}

inline long power( short base, short exponent )
{
long product = 1;
short i;

if ( exponent < 0 )
return( 0 );

for ( i=1; i<=exponent; i++ )
product *= base;

return product;
}
```

The output is as foillows:

```
power( 2, 3 ): 8
power( 3, 6 ): 729
power( 5, 0 ): 1
power( -3, 4 ): 81
```

`inline.cpp` starts with the standard include file, followed by a function prototype that features the keyword `inline`: `inline long power( short base, short exponent )`.

`main()` calls `power()` four times and prints the result of each call.

By preceding `power()`'s declaration by the `inline` keyword, we've asked the compiler to replace each of the four function calls in `main()` with the code in `power()`. Note that this replacement effects the object code and has no impact on the source code:

There are two clear benefits that arise from using `inline` code instead of a `#define macro`

- *type-safety*
- *side-effects protection*

Consider this `#define` macro:

```
#define square(a) ( (a) * (a) )
```

Compare that macro to this inline function:

```
inline int square( int a )
{
return( a * a );
}
```

The inline version restricts its parameter to an integral value while the `#define` performs a simple-minded text substitution.

Now suppose you call `square()` with a prefix operator:

```
xSquared = square( ++x );
```

The `#define` version expands this as follows:

```
xSquared = ( (++x) * (++x) );
```

which has the unwanted side-effect of incrementing `x` twice. The `inline` version doesn't do this. The upshot here is that both `#defines` and `inlines` offer an inline performance advantage, but the `inline` does its job a little more carefully.

Chapter 54

Object Oriented Programming

This chapter introduces concepts of object oriented programming and details how C++ implements objects we see how C++ may be used to create, destroy, and manipulate objects in very powerful ways.

First let's discuss the concept of an object.

54.1 Objects

There is nothing mysterious about the concept of an object. In C++, an object is any instance of a data type. For example, this line of code:

```
int myInt;
```

declares an int object.

The first real object we'll take a look at is the `struct` structure: One of the most valuable features shared by C and C++ is the structure. Without the structure, you'd have no way to group data that belonged together. For example, suppose you wanted to implement an employee data base that tracked an employee's name, employee ID, and salary. You might design a structure that looks like this:

```
const short kMaxNameSize = 20;

struct Employee
 {
   char name[kMaxNameSize];
   long id;
   float salary;
 };
```

The great advantage of this structure is that it lets you bundle several pieces of information together under a single name. This concept is known as encapsulation.

For example, if you wrote a routine to print an employee's data, you could write:

```
Employee newHire;

......

PrintEmployee(newHire.name, newHire.id, newHire.salary);
```

Did you notice anything unusual about the declaration of `newHire` in the preceding code sample?

- In C, this code would not have compiled. Instead, the declaration would have looked like this:

  ```
  struct Employee newHire; /* The C version */
  ```

- When the C++ compiler sees a structure declaration, it uses the structure name to create a new data type, making it available for future structure declarations.

On the other hand, it would be so much more convenient to pass the data in its encapsulated form:

```
PrintEmployee( &newHire );
```

Encapsulation allows you to represent complex information in a more natural, easily accessible form. In the C language, the `struct` is the most sophisticated encapsulation mechanism available.

C++ takes encapsulation to a even higher level: Whilst C structures are limited strictly to data, C++ supports structures composed of both data and functions.

Here's an example of a C++ structure declaration:

```
const short kMaxNameSize = 20;
struct Employee
  {
    // Data members...
    char employeeName[ kMaxNameSize ];
    long employeeID;
    float employeeSalary;

    // Member functions...
    void PrintEmployee();
   };
```

This example declares a new type named `Employee`. You can use the `Employee` type to declare individual `Employee` objects:

- Each `Employee` object is said to be a member of the `Employee` *class*.
- The Employee `class` consists of *three data fields* as well as a *function* named `PrintEmployee()`.
  - In C++, a class's data fields are known as *data members* and its functions are known as *member* functions.
- Each `Employee` object you create gets its own copy of the `Employee` class data members.
- All `Employee` objects share a single set of Employee member functions.

You can also create classes, using the exact same syntax, substituting the keyword class for struct.It is more commom to declare the stucture as a `class`, to keep Object Oriented principles in tact:

```
const short kMaxNameSize = 20;

class Employee
  {
    // Data members...
    char employeeName[ kMaxNameSize ];
    long employeeID;
    float employeeSalary;
    // Member functions...
    void PrintEmployee();
   };
```

The `class` definition is slightly different in how data members and member functions can be accessed which we will address later (See Access Privilege, Section 54.6).

The only difference is, the members of a struct are all `public` by default and the members of a class are all `private` by default.

Why use class instead of struct?

If you start with a struct, you give the world complete access to your class members unless you intentionally limit access using the appropriate access specifiers. If you start with a class, access to your class members is limited right from the start. You have to intentionally allow access by using the appropriate access specifiers.

We will soon be using the `class` for the remainder of this course, only in the next few examples will we use `struct` we'll use the class keyword to declare our classes.

54.2 Encapsulating Data and Functions

Later in this chapter, we'll see how to access an object's data members and member functions. For now, let's take a look at the mechanisms C++ provides to create and destroy objects.

There are two ways to create a new object. The simplest method is to define the object directly, just as you would a local variable:

```
Employee employee1;
```

This definition creates an `Employee` object whose name is `employee1`. `employee1` consists of a block of memory large enough to accommodate each of the three `Employee` data members. When you create an object by defining it directly, as we did above, memory for the object is allocated when the definition moves into scope. That same memory is freed up when the object drops out of scope.

For example, you might define an object at the beginning of a function:

```
void CreateEmployee()
{
Employee employee1;

....
}
```

When the function is called, memory for the object is allocated, right along with the function's other local objects. When the function exits, the object's memory is deallocated. When the memory for an object is deallocated, the object is said to be destroyed.

54.3 Creating an Object

If you want a little more control over when your object is destroyed, use `new` operator:

- First, define an object pointer,
- then call `new` to allocate the memory for your object. `new` returns a pointer to the newly created object.

An example that creates an tt Employee object this way follows:

```
Employee *employeePtr;

employeePtr = new Employee;
```

The first line of code defines a pointer designed to point to an `Employee` object. The second line uses `new` to create an `Employee` object. `new` returns a pointer to the newly created `Employee`.

Once you've created an object, you can modify its data members and call its member functions.

- If you've defined the object directly, you'll refer to its data members using the . operator:

  ```
  Employee employee1;
  employee1.employeeSalary = 200.0;
  ```

- If you're referencing the object through a pointer, use the $->$ operator:

  ```
  Employee *employeePtr;
  employeePtr = new Employee;
  employeePtr->employeeSalary = 200.0;
  ```

- To call a member function, use the same technique. If the object was defined directly, you'll use the . operator:

  ```
  Employee employee1;
  employee1.PrintEmployee();
  ```

- If you're referencing the object through a pointer, you'll use the $->$ operator:

  ```
  Employee *employeePtr;
  employeePtr = new Employee;
  employeePtr->PrintEmployee();
  ```

54.3.1 The Current Object

In the above examples, each reference to a data member or member function started with an object or object pointer. Inside a member function, however, the object or object pointer isn't necessary to refer to the object for which the member function is executing.

For example, inside the `PrintEmployee()` function, you can refer to the data member `employeeSalary` directly, without referring to an object or object pointer:

```
if ( employeeSalary <= 200 )
cout << "Give this person a raise!!!";
```

This code is kind of puzzling. What object does `employeeSalary` belong to? After all, you're used to writing:

```
myObject->employeeSalary
```

instead of just:

```
employeeSalary
```

The key to this puzzle lies in knowing which object spawned the call of `PrintEmployee()` in the first place. Although this may not be obvious, a call to a nonstatic member function must originate with a single object.

As we'll see later, class members may be declared as `static`. A `static` data member holds a value that is global to a class and not specific to a single object of that class. A *static member*function is usually designed to work with a class's static data members.

Suppose you called `PrintEmployee()` from a non-`Employee` function (such as `main()`). You must precede this call with a reference to an object:

```
employeePtr->PrintEmployee();
```

Whenever a member function is called, C++ keeps track of the object used to call the function. This object is known as the current object.

In the call of `PrintEmployee()` above, the object pointed to by `employeePtr` is the current object. Whenever this call of `PrintEmployee()` refers to an Employee data member or function without using an object reference, the current object (in this case, the object pointed to by `employeePtr`) is assumed.

Suppose `PrintEmployee()` then called another `Employee` function. The object pointed to by `employeePtr` is still considered the current object. A reference to `employeeSalary` would still refer to the current object's copy of `employeeSalary`. The point to remember is, a nonstatic member function always starts up with a single object in mind.

54.3.2 The `This` Object Pointer

C++ provides a generic object pointer, available inside any mem-ber function, that points to the current object. The generic pointer has the name `this`. For example, inside every `Employee` function, the line:

```
this->employeeSalary = 400;
```

is equivalent to this line:

```
employeeSalary = 400;
```

`this` is useful when a member function wants to return a pointer to the current object, pass the address of the current object on to another function, or just store the address somewhere. This line of code:

```
return this;
```

returns the address of the current object.

54.4 Destroying an Object

When you create an object using `new`, you've got to take responsibility for destroying the object at the appropriate time. Just as a C programmer balances a call to `malloc()` with a call to `free()`, a C++ programmer balances each use of the `new` operator with an eventual use of the `delete` operator.

Here's the syntax:

```
Employee *employeePtr;

employeePtr = new Employee;
delete employeePtr;
```

As you'd expect, `delete` destroys the specified object, freeing up any memory allocated for the object. Note that this freed up memory only includes memory for the actual object and does not include any extra memory you may have allocated.

For example, suppose the object is a structure and one of its data members is a pointer to another structure. When you delete the first structure, the second structure is not deleted. If delete is used with a pointer having a value of 0, delete does nothing. If the pointer has any other value delete will try to destroy the specified object.

54.5 Member Functions

Once your structure is declared, you're ready to write your member functions. Member functions behave in much the same way as ordinary functions, with a few small differences. One difference, pointed out earlier, is that a member function has access to the data members and member functions of the object used to call it. Another difference lies in the function implementation's title line. Here's a sample:

```
void Employee::PrintEmployee()
{
cout << "Employee Name: " << employeeName << "\n";
}
```

Notice that the function name is preceded by the class name and two colons (`Employee::`)

This notation is **mandatory** and tells the compiler that this function is a member of the specified class.

54.5.1 The Constructor Function

Typically, when you create an object, you'll want to perform some sort of initialization on the object. For instance, you might want to provide initial values for your object's data members.

The *constructor function* is C++'s built-in initialization mechanism. The constructor function (or just plain constructor) is a member function that has the **same name** as the object's class. For example, the constructor for the `Employee` class is named `Employee()`.

When an object is created, the constructor for that class gets called.

Consider this code:

```
Employee *employeePtr;

employeePtr = new Employee;
```

In the second line, the new operator allocates a `new Employee` object, then immediately calls the object's constructor. Once the constructor returns, the address of the new object is assigned to `employeePtr`.

This same scenario holds true in this declaration:

```
Employee employee1;
```

As soon as the object is created, its constructor is called. Here's our `Employee` struct declaration with the constructor declaration added in:

```
const short kMaxNameSize = 20;

struct Employee
```

```
   {
     // Data members...
     char employeeName[ kMaxNameSize ];
     long employeeID;
     float employeeSalary;

     // Member functions...
     Employee();
    void PrintEmployee();
  };
```

Notice that the constructor is declared *without* a return value. Constructors **never** return a value.

Here's a sample constructor:

```
Employee::Employee()
{
employeeSalary = 200.0;
}
```

Constructors are optional. If you don't have any initialization to perform, there is no need to define one.

You can add parameters to your constructor function. Constructor parameters are typically used to provide initial values for the object's data members. Here's a new version of the Employee() constructor:

```
Employee::Employee(char *name, long id, float salary)
{
   strncpy(employeeName, name, kMaxNameSize);
   employeeName[ kMaxNameSize - 1 ] = '\0';
   employeeID = id;
   employeeSalary = salary;
}
```

The constructor copies the three parameter values into the corresponding data members.

Notice that `strncpy()` was used, ensuring that the copy will work, even if the source string was not prop-erly terminated. A `NULL` terminator is provided at the end of the string for just such an emergency.

The object that was just created is always the constructor's current object. In other words, when the constructor refers to an `Employee` data member, such as `employeeName` or `employeeSalary`, it is referring to the copy of that data member in the newly created object.

This line of code supplies the new operator with a set of parameters to pass on to the constructor:

```
employeePtr = new Employee( "David Marshall", 1000, 200.0 );
```

This line of code does the same thing without using new:

```
Employee employee1( "David Marshall", 1000, 200.0 );
```

As you'd expect, this code creates an object named `employee1` by calling the `Employee` constructor, passing it the three specified parameters.

Just for completeness, here's the class declaration again, showing the new constructor:

```
struct Employee
  {
    // Data members...
    char employeeName[ kMaxNameSize ];
    long employeeID;
    float employeeSalary;

    // Member functions...
    Employee( char *name, long id, float salary );
    void PrintEmployee();
};
```

54.5.2 The Destructor Function

The destructor function is called automatically when you `delete` an object or it goes out of scope.

Use the destructor to clean up after your object before it goes away. For instance, you might use the destructor to deallocate any additional memory your object may have allocated.

The destructor function is named by a tilde character (∼) followed by the class name. The destructor for the Employee class is named ∼`Employee()`. The destructor has no return value and no parameters.

Here's a sample destructor:

```
Employee::~Employee()
{
cout << "Deleting employee #" << employeeID << "\n";
}
```

If you created your object using `new`, the destructor is called when you use delete:

```
Employee *employeePtr;

employeePtr = new Employee;
delete employeePtr;
```

If your object was defined directly, the destructor is called just before the object is destroyed. For example, if the object was declared at the beginning of a function, the destructor is called when the function exits.

If your object was declared as a global or static variable, its constructor will be called at the beginning of the program and its destructor will be called just before the program exits. Yes, global objects have scope, just as local objects do.

Here's an updated `Employee` *class* declaration showing the constructor and destructor:

```
struct Employee
 {
   // Data members...
   char employeeName[kMaxNameSize];
   long employeeID;
   float employeeSalary;

   // Member functions...
   Employee( char *name, long id, float salary );
   ~Employee();
   void PrintEmployee();
  };
```

When you declare a `class`, you need to decide which data members and functions you'd like to make available to the rest of your program. C++

gives you the power to hide a class's functions and data from all the other functions in your program, or allow access to a select few.

For example, consider the `Employee` class we've been working with throughout this chapter. In the current model, an Employee's name is stored in a single array of chars. Suppose you wrote some code that created a new Employee, specifying the name, id, and salary, then later in your program you decided to modify the Employee's name, perhaps adding a middle name provided while your program was running.

With the current design, you could access and modify the Employee's `employeeName` data member from anywhere in your program. As time passes and your program becomes more complex, you might find yourself accessing `employeeName` from several places in your code.

Now imagine what happens when you decide to change the implementation of `employeeName`. For example, you might decide to break the single employeeName into three separate data members, one each for the first, middle, and last names. Imagine the hassle of having to pore through your code finding and modifying every single reference to `employeeName`, making sure you adhere to the brand new model.

C++ allows you to hide the implementation details of a class (the specific type of each data member, for example), funneling all access to the implementation through a specific set of interface routines. By hiding the implementation details, the rest of your program is forced to go through the interface routines your class provides. That way, when you change the implementation, all you have to do is make whatever changes are necessary to the class's interface, without having to modify the rest of your program.

54.6 Access Priveleges

The mechanism C++ provides to control access to your class's implementation is called the access specifier.

C++ allows you to assign an *access specifier* to any of a class's data members and member functions. The access specifier defines which of your program's functions have access to the specified data member or function. The access specifier must be `public`, `privat`, or `protected`:

- If a data member or function is marked as `private`, access to it is limited to member functions of the same class (or, as you'll see later in the chapter, to classes or member functions marked as a friend of the class).

- The `public` specifier gives complete access to the member function or data member, limited only by scope.
- The third C++ access code is `protected`. The protected access code offers the same protection as private, with one exception. A `protected` data member or function can also be accessed by a class derived from the current class. Since we won't get to derived classes till later in the book, we'll put off discussion of the protected access code till then.

By default, the data members and member functions of a class declared using the `struct` keyword are all `public`. By adding the `private` keyword to our `class` declaration, we can limit access to the Employee data members, forcing the outside world to go through the provided member functions:

```
struct Employee
{
   // Data members...
   private:
     char employeeName[ kMaxNameSize ];
     long employeeID;
     float employeeSalary;

   // Member functions...
   public:
     Employee( char *name, long id, float salary );
     ~Employee();
     void PrintEmployee();
};
```

Once the compiler encounters an access specifier, all data members and functions that follow are marked with that code, at least until another code is encountered. In this example, the three data members are marked as private and the three member functions are marked as public.

Note the : after the access specifier. Without it, your code won't compile!

Here's the new version of the Employee `class`:

class Employee // Data members... private: char employeeName[kMaxNameSize]; long employeeID; float employeeSalary;

// Member functions... public: Employee(char *name, long id, float salary); Employee(); void PrintEmployee(); ;

Notice that the private access specifier is still in place. Since the members of a class-based class are private by default, the private access specifier is not needed here, but it does make the code a little easier to read. The public access specifier is necessary, however, to give the rest of the program access to the Employee member functions.

54.7 employee.cpp, source code

We now give a complete code listing to bring togther all the facets we have met in this chapter so far.

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

const short kMaxNameSize = 20;

class Employee
{
// Data members...
private:
char employeeName[ kMaxNameSize ];
long employeeID;
float employeeSalary;

// Member functions...
public:
Employee( char *name, long id, float salary );
~Employee();
void PrintEmployee();
};

Employee::Employee( char *name, long id, float salary )
```

```
{
strncpy( employeeName, name, kMaxNameSize );

employeeName[ kMaxNameSize - 1 ] = '\0';

employeeID = id;
employeeSalary = salary;

cout << "Creating employee #" << employeeID << "\n";
}

Employee::~Employee()
{
cout << "Destroying employee #" << employeeID << "\n";
}

void Employee::PrintEmployee()
{
cout << "-----\n";
cout << "Name:   " << employeeName << "\n";
cout << "ID:     " << employeeID << "\n";
cout << "Salary: " << employeeSalary << "\n";
cout << "-----\n";
}

int main()
{
Employee employee1( "Dave Mark", 1, 200.0 );
Employee *employee2;

employee2 = new Employee( "Steve Baker", 2, 300.0 );

employee1.PrintEmployee();
employee2->PrintEmployee();

delete employee2;

return 0;
}
```

54.8 Friends

In our last program, the `Employee` class marked its data members as private and its member functions as public. As we discussed earlier, the idea behind this strategy is to hide the implementation details of a class from the rest of the program, funneling all access to the class's data members through a set of interface routines.

For example, suppose we wanted to provide access to the Employee class's `employeeSalary` data member. Since `employeeSalary` is marked as private, there's no way to access this data member outside the Employee class. If we wanted to, we could provide a pair of public member functions a user of the `Employee` class could use to retrieve (`GetEmployeeSalary()`) and modify (`ChangeEmployeeSalary()`) the value of employeeSalary.

Sometimes this strategy just doesn't work well: For example, suppose you created a Payroll class to generate paychecks for your Employees. Clearly, the Payroll class is going to need access to an Employee 's salary.

But if you create a public `GetEmployeeSalary()` member function (or mark `employeeSalary` as public) you'll make `employeeSalary` available to the entire program, something you might not want to do.

The solution to this problem is provided by C++'s *friend mechanism.* C++ allows you to designate a class or a single member function as a *friend* to a specific class. In the previous example, we could designate the Payroll class as a friend to the Employee class:

```
//-------------------------------------- Payroll

class Payroll
{
// Data members...
private:

// Member functions...
public:
Payroll();
~Payroll();
void PrintCheck( Employee *payee );
};

//-------------------------------------- Employee

class Employee
{
 friend class Payroll;

  // Data members...
  private:
   char employeeName[ kMaxNameSize ];
   long employeeID;
  float employeeSalary;

 // Member functions...
  public:
  Employee( char *name, long id, float salary );
  ~Employee();
    void PrintEmployee();
};
```

The `friend` statement, in the first line of the Employee class declaration, is *always* placed in the class whose data members and functions are being shared. In this case, the `Employee` class is willing to share its private data members and functions with its new friend, the `Payroll` class. Once the `Payroll` class has friend access to the Employee class, it can access private data members and functions like `employeeSalary`.

54.8.1 Three Types of Friends

There are three ways to designate a friend.

- (As we've already seen) You can designate an entire class as a friend to a second class.
- You can also designate a specific class function as a friend to a class. For example, the Payroll class we just declared contains a function

named `PrintCheck()`. We might want to designate the `PrintCheck()` function as a friend of the Employee class, rather than the entire Payroll class.

```
class Employee
 {
   friend void Payroll::PrintCheck( Employee *payee );;

   // Data members...
   private:
     char employeeName[ kMaxNameSize ];
     long employeeID;
     float employeeSalary;

   // Member functions...
   public:
     Employee( char *name, long id, float salary );
     ~Employee();
     void PrintEmployee();
  };
```

This time, the friend definition specified the Payroll mem-ber function `Payroll::PrintCheck()` instead of the entire Payroll class. Since the friend statement referred to a member function of another class, the full name of the function (including the class name and the two colons) was included.

- You can also designate a nonmember function as a friend. For example, you could designate `main()` as a friend to the Employee class:

```
class Employee
 {
   friend int main();

   // Data members...
   private:
     char employeeName[ kMaxNameSize ];
     long employeeID;
```

```
          float employeeSalary;

        // Member functions...
        public:
          Employee( char *name, long id, float salary );
          ~Employee();
          void PrintEmployee();
      };
```

This arrangement gives main() access to all Employee data members and functions, even those marked as private. Just because main() is a friend doesn't give any special priveleges to any other functions, however. Choose your friends carefully!

54.8.2 A Friendly Example

Our next example combines the Employee class created earlier with the Payroll class described in this section.

The `friends.cpp` source code follows:

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

const short kMaxNameSize = 20;

class Employee;


//---------------------------------------  Payroll

class Payroll
{
// Data members...
private:

// Member functions...
public:
Payroll();
```

```
~Payroll();
void PrintCheck( Employee *payee );
};

//--------------------------------------  Employee

class Employee
{
friend void Payroll::PrintCheck( Employee *payee );

// Data members...
private:
char employeeName[ kMaxNameSize ];
long employeeID;
float employeeSalary;

// Member functions...
public:
Employee( char *name, long id, float salary );
~Employee();
void PrintEmployee();
};


//-------------------  Payroll Member Functions

Payroll::Payroll()
{
cout << "Creating payroll object\n";
}

Payroll::~Payroll()
{
cout << "Destroying payroll object\n";
}

void Payroll::PrintCheck( Employee *payee )
{
```

```
cout << "Pay $" << payee->employeeSalary
 << " to the order of "
 << payee->employeeName << "...\n\n";
}


//------------------- Employee Member Functions

Employee::Employee( char *name, long id, float salary )
{
strncpy( employeeName, name, kMaxNameSize );

employeeName[ kMaxNameSize - 1 ] = '\0';

employeeID = id;
employeeSalary = salary;

cout << "Creating employee #" << employeeID << "\n";
}

Employee::~Employee()
{
cout << "Destroying employee #" << employeeID << "\n";
}

void Employee::PrintEmployee()
{
cout << "-----\n";
cout << "Name:   " << employeeName << "\n";
cout << "ID:     " << employeeID << "\n";
cout << "Salary: " << employeeSalary << "\n";
cout << "-----\n";
}


//-------------------------------------- main

int main()
```

```
{
Employee *employee1Ptr;
Payroll *payroll1Ptr;

payroll1Ptr = new Payroll;

employee1Ptr = new Employee( "Carlos Derr", 1000, 500.0 );

employee1Ptr->PrintEmployee();

payroll1Ptr->PrintCheck( employee1Ptr );

delete employee1Ptr;
delete payroll1Ptr;

return 0;
}
```

The Output looks like this:

```
Creating payroll object
Creating employee #1000
----
Name: Carlos Derr
ID: 1000
Salary: 500
----
Pay $500 to the order of Carlos Derr...
Destroying employee #1000
Destroying payroll object
```

`friends.cpp` starts out just like `employee.cp`, with the same two `#include`s and the same `const` definition:

Since the Payroll class declaration refers to the Employee class (check out the parameter to `PrintCheck()`) and its declaration comes first, we'll need a forward declaration of the Employee

Next comes the declaration of the Payroll class. To keep this example as simple as possible, we've stripped Payroll down to its bones: no data members, a constructor, a destructor, and a `PrintCheck()` function. Further down in the source, the Employee class will mark the `PrintCheck()` function as a friend. Next comes the Employee class declaration. You may have noticed that we didn't list the Payroll member functions right after the Payroll class declaration. This was because `Payroll::PrintCheck()` refers to the Employee data mem-ber employeeSalary, which hasn't been declared yet.

Take a look at the friend declaration inside the Employee class declaration. Notice that we've opted to make `Payroll::PrintCheck()` a friend of the Employee class. Now, `PrintCheck()` is the only Payroll function with access to the private Employee data members.

Interestingly, if you leave `PrintCheck()`'s parameter out of the friend statement, the code won't compile. Since you can have more than one function with the same name (Recall overloaded functions), if the parameter is left out, the compiler tries to match the friend statement with a version of `PrintCheck()` with no parameters. When it doesn't find one, the compiler reports an error.

Next come the Payroll member functions. The constructor and destructor print messages letting you know they were called, while `PrintCheck()` prints up a simulated check using the private Employee data members employeeSalary and employeeName.

The Employee member functions are the same as they were in declared next.

Once again, `main()` is where the action is. We start off by defining a couple of pointers, one to an Employee object and one to a Payroll object.

The two constructors are called and then `PrintEmployee()` is called.

Next, `PrintCheck()` is called. `PrintCheck()` takes a pointer to the Employee object as a parameter. A check is printed to the specified Employee using employeeName and employeeSalary:

Finally, both objects are `deleted`: The two destructors functions calles which print their respective messages.

Chapter 55

Inheritance, Derived Functions, Virtual Functions

C++ allow you to use one class dclaration, known as a *base class*, as the basis for the declaration of a second class, known as a *derived class*.

For example, you might declare an Employee class that describes your company's employees. Next, you might declare a Sales class, based on the Employee class, that describes employees in the sales department.

This chapter is emphasize the advantages of classes derived from other classes and gives several examples.

55.1 Inheritance

One of the most important features of class derivation is *inheritance*. A derived class inherits *all* of the data members and member functions from its base class.

As an example, consider the following class declaration:

```
class Base
{
  public:
   short baseMember;
   void SetBaseMember( short baseValue );
};
```

This class, Base, has two members, a data member named baseMember and a member function named SetBaseMember().

Both of these members will be inherited by any classes derived from this class.

Here's another class declaration:

```
class Derived : Base
{
public:
short derivedMember;
void SetDerivedMember( short derivedValue );
}
```

This class is a derived class named, appropriately enough, Derived. The : Base at the end of the title tells you that this object is derived from the class named Base.

As you'd expect, this object has its own copy of the data member derivedMember as well as access to the member function SetDerivedMember().

What you might not have expected are the members inherited by this object, that is, the Base class data member baseMember as well the Base class member function SetBaseMember().

Here's some code that allocates a Derived object, then accesses various data members and functions:

```
Derived *derivedPtr;

derivedPtr = new Derived;
derivedPtr->SetDerivedMember( 20 );
cout << "derivedMember = " << derivedPtr->derivedMember;
derivedPtr->SetBaseMember( 20 );
cout << "\nbaseMember = " << derivedPtr->baseMember;
```

Notice that the object pointer derivedPtr is used to access its own data members and functions as well as its inherited data members and functions. Notice also that the example does not create a Base object. This is important. When an object inherits data members and functions from its base class, the compiler allocates the extra memory needed for all inherited members right along with memory for the object's own members. class is derived from the class named Base. As you'd expect, this object has its own copy of the data member derivedMember as well as access to the member function

55.2 Access and Inheritance

Although a derived class inherits all of the data members and member functions from its base class, it doesn't necessarily retain access to each member.

Here's how this works. When you declare a derived class, you declare its base class as either `public` or `private`. One way to do this is to include either `public` or `private` in the title line of the declared class.

For example, in the declaration below, the class `Base` is marked as `public`:

```
class Derived : public Base
{
   public:
     short derivedMember;
     void SetDerivedMember( short derivedValue );
}
```

In the next declaration, `Base` is marked as private:

```
class Derived : private {\tt Base}
{
public:
short derivedMember;
void SetDerivedMember( short derivedValue );
}
```

You can also mark a base class as `public` or `private` by leaving off the access specifier. If you use the `class` keyword to declare the derived class, the base class defaults to `private`. If you use the `struct` keyword to declare the derived class, the base class defaults to `public`.

Once you know whether the base class is public or private, you can determine the access of each of its inherited members by following these three rules:

- The derived class does not have access to private members inherited from the base class. This is true regardless of whether the base class is public or private.
- If the base class is public, the members inherited from the base class retain their access level (providing the inherited member is not private, of course). This means that an inherited public member remains public and an inherited protected member remains protected.

- If the base class is private, the members inherited from the base class are marked as private in the derived class.

Previously, we adopted the strategy of declaring our data members as private and our member functions as public. This approach works well if the class will never be used as a base class for later derivation. If you ever plan on using a class as the basis for other classes, declare your data members as protected and your member functions as public.

A protected member can be accessed only by members of its class or by members of any classes derived from its class. In a base class, protected is just like private. The advantage of protected is that it allows a derived class to access the member, while protecting it from the outside world. We'll get back to this strategy in a bit. For now, let's take a look at an example of class derivation.

55.3 A Class Derivation Example

So far in this chapter, we've learned how to derive one class from another and you've been introduced to the protected access specifier. Our first sample program brings these lessons to life.

```
#include <iostream.h>


//---------------------------------------  Base

class Base
{
// Data members...
private:
short baseMember;

// Member functions...
protected:
void SetBaseMember( short baseValue );
short GetBaseMember();
};
```

```
void Base::SetBaseMember( short baseValue )
{
baseMember = baseValue;
}

short Base::GetBaseMember()
{
return baseMember;
}


//--------------------------------------- Base:Derived

class Derived : public Base
{
// Data members...
private:
short derivedMember;

// Member functions...
public:
void SetMembers( short baseValue,
short derivedValue );
void PrintDataMembers();
};

void Derived::SetMembers( short baseValue,
short derivedValue )
{
derivedMember = derivedValue;
SetBaseMember( baseValue );
}

void Derived::PrintDataMembers()
{
cout << "baseMember was set to "
<< GetBaseMember() << '\n';
cout << "derivedMember was set to "
<< derivedMember << '\n';
}


//-------------------------------------- main()

int main()
{
Derived *derivedPtr;

derivedPtr = new Derived;

derivedPtr->SetMembers( 10, 20 );

derivedPtr->PrintDataMembers();

return 0;
}
```

The ouput of this program is:

```
baseMember was set to 10
derivedMember was set to 20
```

Let's take a look at the source code.

As usual, `derived.cpp` starts by including <iostream.h>:

Next, we declare a class named `Base`, which we'll use later as the basis for a second class named `Derived`. Base has a single data member, a short named `baseMember`, which is marked as `private`.

Base also includes two member functions, each marked as `protected`. `SetBaseMember()` sets baseMember to the specified value while `GetBaseMember()` returns the current value of `baseMember`.

Since `baseMember is private`, it cannot be accessed by any function outside the `Base` class. Since `SetBaseMember()` and `GetBaseMember()` are protected, they can only be accessed by `Base` member functions and from within any classes derived from `Base`. Note that `main()` cannot access either of these functions.

Our second class, `Derived`, is derived from Base:

The `public` keyword following the colon in the class title line marks `Base` as a public base class. As mentioned earlier, if a base class is declared as public, all inherited public members remain `public` and inherited `protected` members remain protected. Inherited `private` members are **not** accessible by the derived class.

If you marked `Base` as `private` instead of `public` all inherited members would be marked as `private` and would not be accessible by any classes derived from Derived. The point here is this:

- If you mark the base class as `private` you effectively **end** the inheritance chain.

As a general rule,*you should declare your derived classes using* `public` *inheritance*:

It's rare that you'd want to reduce the amount of information inherited by a derived class. Most of the time, a derived class is created to extend the reach of the base class by adding new data members and functions.

`Derived` has a single data member, a short named `derivedMember`, which is declared as private. `derivedMember` can only be accessed by a `Derived` member function. Derived contains two member functions, `SetMembers()` and `PrintDataMembers()`.

`SetMembers()` takes two shorts, assigns the first to `derivedMember`, and passes the second to `SetBaseMember()`. `SetBaseMember()` was used because `Derived` does not have direct access to `baseMember`.

`PrintDataMembers()` prints the values of `baseMember` and `derivedMember`. Since `Derived` doesn't have direct access to `baseMemeber`, `GetBaseMember()` is called to retrieve the value.

`main()` starts by declaring a `Derived` pointer and then using new to create a `new Derived` object (since we didn't include a constructor for either of our classes, nothing exciting has hap-pened yet):

It's important to understand that when the `Derived object` is created, it receives its own copy of `baseMember`, even though it doesn't have access to baseMember. If the `Derived` object wants to modify its copy of `baseMember`, it will have to do so via a call to `Base::SetBaseMember()`, which it also inherited. Now things start to get interesting. `main()` uses the pointer to the Derived object to call `SetMembers()`, setting its copy of `baseMember` to 10 and `derivedMember` to 20.

Next, we call the `Derived` member function `PrintDataMembers()`

The values of the two data members are successfully set when the program is ran:

`baseMember` was set to 10;

`derivedMember` was set to 20,

Just as the `Derived` object pointer is able to take advantage of inheritance to call `SetBaseMember()`, `PrintDataMembers()` is able to print the value of the inherited data member `baseMember` by calling `GetBaseMember()`.

55.4 Derivation, Constructors and Destructors

When an object is created, its constructor is called to initialize the object's data members. When the object is deleted, its destructor is called to perform any necessary cleanup.

Suppose the object belongs to a derived class, and suppose it inherits a few data members from its base class.

How do these inherited data members get initialized?

When the object is deleted, who does the cleanup for the inherited data members?

As it turns out, C++ solves this tricky issue for you. Before the compiler calls an object's constructor, it first checks to see whether the object belongs to a derived class. If so, the constructor belonging to the base class is called and then the object's own constructor is called. The base class constructor initializes the object's inherited members, while the object's own constructor initializes the members belonging to the object's class .

The reverse holds true for the destructor. When an object of a derived class is deleted, the derived class's destructor is called and then the base class's destructor is called.

55.4.1 The Derivation Chain

There will frequently be times when you derive a class from a base class that is, itself, derived from some other class. Each of these classes acts like a link in a derivation chain. The constructor/ destructor calling sequence just described still holds, no matter how long the derivation chain.

Suppose you declare three classes, A, B, and C, where class B is derived from A and C is derived from B.

When you create an object of class B, it will inherit the mem-bers from class A. When you create an object of class C, it will inherit the members from class B, which includes the previously inherited members from class A.

When an object from class C is created, the compiler follows the derivation chain from C to B to A and discovers that A is the ultimate base class in this chain. The compiler calls the class A constructor, then the class B constructor, and finally the class C constructor.

When the object is deleted, the class C destructor is called first, followed by the class B destructor and, finally, by the class A destructor.

55.4.2 A Derivation Chain Example

Our second sample program demonstrates the order of constructor and destructor calls in a three-class derivation chain. Close the current project by selecting from the menu.

The source code listing is:

```
#include <iostream.h>


//--------------------------------------  Gramps

class Gramps
{
// Data members...

// Member functions...
public:
Gramps();
~Gramps();
};

Gramps::Gramps()
{
cout << "Gramps' constructor was called!\n";
}
```

```
Gramps::~Gramps()
{
cout << "Gramps' destructor was called!\n";
}


//---------------------------------  Pops:Gramps

class Pops : public Gramps
{
// Data members...

// Member functions...
public:
Pops();
~Pops();
};

Pops::Pops()
{
cout << "Pops' constructor was called!\n";
}

Pops::~Pops()
{
cout << "Pops' destructor was called!\n";
}


//---------------------------------  Junior:Pops

class Junior : public Pops
{
// Data members...

// Member functions...
public:
```

```
Junior();
~Junior();
};

Junior::Junior()
{
cout << "Junior's constructor was called!\n";
}

Junior::~Junior()
{
cout << "Junior's destructor was called!\n";
}


//--------------------------------------- main

int main()
{
Junior *juniorPtr;

juniorPtr = new Junior;

cout << "----\n";

delete juniorPtr;

return 0;
}
```

The output of the program is as follows:

```
Gramps' constructor was called!
Pops' constructor was called!
Junior's constructor was called!
----
Junior's destructor was called!
Pops' destructor was called!
Gramps' destructor was called!
```

As you can see by the output, each class constructor was called once, then each class destructor was called once in reverse order.

Notice that none of the classes in this program have any data members. For the moment, we're interested only in the order of constructor and destructor calls. Both the constructor and the destructor are declared `public`. The `Gramps` constructor and destructor are pretty simple; each prints an appropriate message to the console:

Our next class, `pops`, is derived from the `Gramps` class. Notice that we use the `public` keyword in the `class` title line. This ensures that the constructor and the destructor inherited from `Gramps` are marked as `public` inside the `Pops` class. Once again, this class has no data members. Both the constructor and the destructor are marked as `public`. They'll be inherited by any class derived from `Pops`.

Just like those of `Gramps`, the `Pops` constructor and destructor are simple and to the point; each sends an appropriate message to the console.

`Junior` is to P͡ops what `Pops` is to `Gramps`. `Junior` inherits not only the `Pops` members but the `Gramps` members as well (as you'll see in a minute, when you create and then delete a `Junior` object, both the `Gramps` and the `Pops` constructor and destructor will be called)

The `Junior` constructor and destructor are just like those of `Gramps` and `Pops`; each sends an appropriate message to the console.

`main()`'s job is to create and delete a single `Junior` object. When the `Junior` object is created, the derivation chain is followed backward until the ultimate base class, `Gramps`, is reached. The `Gramps` constructor is called, giving the `Gramps` class a chance to initialize its data members. Next, the `Pops` constructor is called, and the `Junior` constructor is called.

Next, the `Junior` object is deleted, and, this time, the derivation chain is followed in the reverse order. The `Junior` destructor is called, then the `Pops` destructor, and, finally, the `Gramps` destructor is called.

Now consider this examle further:

Notice in the source code listing above that each of the three classes marked their constructor and destructor as `public`. What would happen if you changed the Gramps constructor to `private`?

Your code wouild not compile because `Junior` no longer has access to the `Gramps` constructor. Now if we change the Gramps constructor from `private` to `protected` the program will compile.

Why?

This time, when `Junior` inherited the `Gramps` constructor it had access to the `Gramps` constructor. Recall that when a derived class inherits a `protected` member from a `public` base class, the inherited member is marked as `protected`.

Now what happens if we change the `Junior` constructor from `public` to `protected` and recompile?

This time the compiler complains that the `Junior` constructor was not accessible. Since the `Junior` constructor was declared `protected` it is accessible by classes derived from `Junior`, but not by outside functions like `main()`.

When `main()` creates a new `Junior` object, it must have access to the `Junior` constructor. On the other hand, you've seen that `main()` does not need access to the `Gramps` or `Pops` constructors to create a `Junior` object. When you changed the `Gramps` constructor to `protected`, `Junior` had access to the `Gramps` constructor and `main()` didn't yet the program still compiles.

55.5 Base Classes and Constructors with Parameters

Our first program in this chapter, `derived`, declared two classes, `Base and Derived`. Neither of these classes included a constructor. Our second program, `gramps`, featured three classes. Though all three classes declared a constructor, none of the constructors declared any parameters.

Our next example enters uncharted waters by declaring classes whose constructors contain parameters.

When are constructor parameters important?

In a world without class derivation, not much. When you start working with derived classes, however, things get a bit more complex.

Consider a base class whose constructor has only a single parameter:

```
class Base
 {
   public:
     Base(short baseParam);
  };
```

Now, add a derived class based on this base class:

```
class Derived : public Base
 {
   public:
     Derived();
 };
```

Notice that the derived class constructor is declared without a parameter. When a `Derived` object is created, the Base constructor is called. What parameter is passed to this constructor?

The secret lies in the definition of the derived class constructor. When a base class constructor has parameters, you have to provide some extra information in the derived class constructor's title line. This information tells the compiler how to map data from the derived class constructor to the base class constructor's parameter list.

For example, we might define the derived class constructor this way:

```
Derived::Derived() : Base(20)
{
cout << "Inside the Derived constructor";
}
```

Notice the `:  Base( 20 )` at the end of the title line. This code tells the compiler to pass the number 20 as a parameter when the Base constructor is called.

This technique is really useful when your derived class constructor also has parameters. Check out the following piece of code:

```
Derived::Derived(short derivedParam) : Base(derivedParam)
{
}
```

This constructor takes a single parameter, `derivedParam`, and maps it to the single parameter in its base class constructor. When a Derived object is created, as follows,

```
Derived *derivedPtr;
derivedPtr = new Derived(20);
```

the parameter is passed to the `Base` constructor. Once the `Base` constructor returns, the same parameter is passed to the `Derived` constructor.

In the preceding example, the `Derived` constructor does nothing but pass along a parameter to the `Base` constructor. Though it may take some getting used to, this technique is quite legitimate. It is perfectly fine to define an empty function whose sole purpose is to map a parameter to a base class constructor.

Class Derivation Example

Our next example program combines the class derivation techniques from our first two programs with the constructor parameter-mapping mechanism described in the previous section.

The code listing, `square.cpp` is as follows:

```
#include <iostream.h>


//--------------------------------------- Rectangle

class Rectangle
{
// Data members...
protected:
short height;
short width;

// Member functions...
public:
Rectangle( short heightParam, short widthParam );
void DisplayArea();
};

Rectangle::Rectangle( short heightParam, short widthParam )
{
height = heightParam;
width = widthParam;
```

```
}

void Rectangle::DisplayArea()
{
cout << "Area is: " <<
height * width << '\n';
}


//---------------------------------------  Rectangle:Square

class Square : public Rectangle
{
// Data members...

// Member functions...
public:
Square( short side );
};

Square::Square( short side ) : Rectangle( side, side )
{
}


//---------------------------------------  main()

int main()
{
Square *mySquare;
Rectangle *myRectangle;

mySquare = new Square( 10 );
mySquare->DisplayArea();

myRectangle = new Rectangle( 10, 15 );
myRectangle->DisplayArea();
```

```
return 0;
}
```

The output of the program is:

```
Area is: 100
Area is: 150
```

square.cp starts in the usual way, by including <iostream.h>:

Next, the first of two classes is declared. `Rectangle` will act as a base class The data members of our base class are declared as `protected`; the member functions, `public`. `height` and `width` hold the height and width of a Rectangle object:

The `Rectangle()` constructor takes two parameters, `heightParam` and `widthParam`, that are used to initialize the `Rectangle` data members.

The member function `DisplayArea()` displays the area of the current object:

The `Square` class is derived from the `Rectangle` class. Just as (geometrically speaking) a square is a specialized form of rectangle (a rectangle whose sides are all equal), a `Square` object is a specialized `Rectangle` object.

The `Square` class has no data members, just a single member function, the `Square()` constructorw hich has one purpose in life — It maps the single `Square()` parameter to the two parameters required by the `Rectangle()` constructor. A square whose side has a length of side is equivalent to a rectangle with a height of side and a width of side:

This is expressed by:

```
Square::Square( short side ) : Rectangle( side, side )
{
}
```

`main()` starts by declaring a Square pointer and a Rectangle pointer. The Square pointer is used to create a new Square object with a side of 10:

As specified by the `Square()` constructor's title line, the compiler calls `Rectangle()`, passing 10 as both `heightParam` and `widthParam`. The `Rectangle()` constructor initializes the Square object's inherited data members `height` and `width` to 10, just as if you'd created a `Rectangle` with a height of 10 and a width of 10.

Next, the `Square` object's inherited member function, `DisplayArea()`, is called and this uses the inherited data members `height` and `width` to calculate the area of the Square (100 in this case.

As far as `DisplayArea()` is concerned, the object whose area it just calculated was a `Rectangle`. It had no idea it was working with data members inherited from a `Rectangle`. This illustrates part of the power of object programming.

Finally, the `Rectangle` pointer is used to create a new `Rectangle` object with a `height` of 10 and a `width` of 15: When we call `DisplayArea()` this time, it displays the appropriate area of 150.

This program demonstrates a *very important point.*

- With just a few lines of code, we can add a new dimension to an existing class without modifying the existing class.

The `Square` class takes advantage of what's already in place, building on the data members and member functions of its base class. Essentially, `Square` added a shortcut to the `Rectangle` class — a quicker way to create a `Rectangle` when the height and width are the same.

This may not seem like a significant gain to you, but there's an important lesson behind this example. C++ makes it easy to build upon existing models, to add functionality to your software by deriving from existing classes.

As you gain experience in object programming, you'll build up a library of classes that you'll use again and again. Sometimes, you'll use the classes as it is. At other times, you'll extend an existing class by deriving a new class from it. By deriving new classes from existing classes, you get the best of both worlds. Code that depends on the base classes will continue to work quite well without modification. Code that takes advantage of the new, derived classes will work just as well, allowing these classes to live in harmony with their base classes.

This examle serves to illustrate is what object programming is all about.

55.6 Overriding Member Function

In the preceding example, the derived class, `Square`, inherited the member function, `DisplayArea()`, from its base class, `Rectangle`. Sometimes, it is useful to *override* a member function from the base class with a more appropriate function in the derived class.

For example, you could have provided Square with its own version of `DisplayArea()` that based its area calculation on the fact that the height and width of a square are equal.

Let us consider another example: Suppose you create a base class named `Shape` and a series of derived classes such as `Rectangle`, `Circle`, and `Triangle`. You can create a `DisplayArea()` function for the Shape class, then override `DisplayArea()` in each of the derived classes.

Suppose you want to create a linked list of `Shape`s. To simplify matters for the software that manages the linked list, you can treat the derived objects as `Shape`s, no matter what their actual type. Then, when you call the `Shape`'s `DisplayArea()` function, their true identity will emerge. A `Triangle` will override the `Shape DisplayArea()` function with a `Triangle DisplayArea()` function. The `Rectangle` and `Circle` will have their own versions as well.

The trick is to get `C++` to call the proper overriding function, if one exists.

55.6.1 Creating a Virtual Function

The `Shape` linked list example we are developing has presented us with a slight problem:

Suppose the linked list contains a pointer to a `Shape` which is actually one of `Rectangle, Triangle`, or `Circle`. Now suppose that `Shape` pointer is used to call the member function `DisplayArea()` like:

```
myShapePtr->DisplayArea();
```

As expressed currently, this code will call the function `Shape::DisplayArea()` even if the Shape pointed to by `myShapePtr` is a `Rectangle, Triangle`, or `Circle`.

The solution to this problem:

Declare `Shape::DisplayArea()` as a *virtual function.*

To do this simply prefix the member function with the `virtual keyword`, in this example:

```
class Shape
{
// Data members...
// Member functions...
public:
virtual void WhatAmI();
};
```

By declaring a base member function as `virtual`, we are asking the compiler to call the overriding function instead of the base function, even if the object used to call the function belongs to the base class.

55.6.2 A Virtual Function Example

Let us now complete this `Shape` example of virtual function overriding. We will be using a base class named `Shape` and two derived classes, `Rectangle` and `Triangle`.

The complete listing for `shape.cpp` is as follows:

```
#include <iostream.h>


//---------------------------------------  Shape

class Shape
{
// Data members...

// Member functions...
public:
virtual void WhatAmI();
};

void Shape::WhatAmI()
{
cout << "I don't know what kind of shape I am!\n";
}


//---------------------------------------  Shape:Rectangle
```

```
class Rectangle : public Shape
{
// Data members...

// Member functions...
public:
void WhatAmI();
};

void Rectangle::WhatAmI()
{
cout << "I'm a rectangle!\n";
}


//--------------------------------------  Shape:Triangle

class Triangle : public Shape
{
// Data members...

// Member functions...
public:
void WhatAmI();
};

void Triangle::WhatAmI()
{
cout << "I'm a triangle!\n";
}


//--------------------------------------  main()

int main()
{
Shape *s1, *s2, *s3;
s1 = new Rectangle;
s2 = new Triangle;
s3 = new Shape;

s1->WhatAmI();
s2->WhatAmI();
s3->WhatAmI();

return 0;
}
```

The output is relatively obvious:

```
I'm a rectangle!
I'm a triangle!
I don't know what kind of shape I am!
```

Initially, the base class `Shape` is declared. `Shape` contains a single member function, `WhatAmI()`. When it is called, `WhatAmI()` tells you what kind of shape it belongs to. Notice that it is declared using the `virtual` keyword, which tells the compiler that you'd like any overriding function to be called, if one exists.

Notice in the actual definition of `Shape::WhatAmI()` that the virtual keyword isn't used here. The virtual keyword is *only allowed* in the function declaration inside the class declaration.

Our next class, `Rectangle`, is derived from the `Shape` class and also has a single member function named `WhatAmI()`: `Rectangle`'s version of `WhatAmI()` is called when the object doing the calling is a `Rectangle` and simply outputs "`I'm a rectangle!`"

The final class, `Triangle`, is also derived from `Shape`, and, once again, `Triangle` has its own version of `WhatAmI()`

`main()` declares three Shape point-ers, `s1, s2`, and `s3`:

Each of these pointers is used to create a new object, a `Rectangle`, a `Triangle`, and a `Shape`, respectively:

```
s1 = new Rectangle;
s2 = new Triangle;
s3 = new Shape;
```

You may be wondering why the three pointers are all declared as `Shape` while the objects assigned to the pointers are of three different types. This is intentional and normal.

If you're building a linked list of shapes, you can store a pointer to each object in the list as a `Shape` pointer rather than as a `Rectangle` pointer or `Triangle` pointer. In this way, your list management software doesn't have to know what type of shape it is dealing with and is much easier to deal with:

If you want to call `WhatAmI()` (or some other, more useful function) for each object in the list, you just step through the list, one object at a time, treating each object as if it were a `Shape`. If the object belongs to a derived class that overrides the function, C++ will make sure the correct function is called.

Once our three objects are created, we try using each object to call `WhatAmI()`:

```
s1->WhatAmI();
s2->WhatAmI();
s3->WhatAmI();
\end
{verbatim}

When the {\tt Rectangle} object ({\tt s1}) is used to call {\tt
WhatAmI()}, we get this result:

\begin{verbatim}
I'm a rectangle!
```

When the `Triangle` object (`s12`) is used to call `WhatAmI()`, we get this result:

```
I'm a triangle!
```

Finally, when the `Shape` object (`s3`) is used to call `WhatAmI()`, we get this result:

```
I don't know what kind of shape I am!
```

In this example, the `Shape`class exists just so that we can create useful, derived classes from it. Creating a `Shape` object is not particularly useful.

55.7 Exercises

Exercise 55.1 *Modify the* `shape.cpp` *prgram to be a true linked list implementation of* `Shape`*s.*

Exercise 55.2 *Modify the* `shape.cpp` *prgram to include a* `Circle` *derived class.*

Chapter 56

Operator Overloading

Operator overloading is another extremely powerful and useful feature of Object Oriented programming

56.1 Overiding Built-in Operators

In C++, you can even overload any of the built-in operators, such as + or * to suit paricular applications.

To see why this might be useful consider the following example:

Imagine that you're running a restaurant and you want to write a program to handle your billing, print your menus, and so on. Your program might create a `MenuItem` class that looks something like this:

```
class MenuItem
{
   private:
     float price;
     char name[ 40 ];

    public:
     MenuItem::MenuItem( float itemPrice, char *itemName );
     float MenuItem::GetPrice( void );
};
```

Your program could define a `MenuItem` object for each item on the menu. When someone orders, you'd calculate the bill by adding together the price of each `MenuItem` like this:

```
MenuItem chicken( 8.99, "Chicken Jalfrezi" );
MenuItem wine( 2.99, "Rioja" );

float total;

total = chicken->GetPrice() + wine->GetPrice();
```

This particular diner had the chicken and a glass of wine. The total is calculated using the member function `GetPrice()`. *Nothing new here*, yet.

Operator overloading provides an alternative way of totalling up the bill. If we program things properly, the compiler will interpret the statement

```
total = chicken + wine;
```

by adding the price of chicken to the price of wine.

Note: Currently, the compiler would complain if you tried to use a *non-integral* type with the *+ operator*.

In C++, You can "reprogram" this operation by giving the + operator a new meaning. To do this, we need to create a function to *overload* the + operator:

```
float operator+( MenuItem item1, MenuItem item2 )
{
return( item1.GetPrice() + item2.GetPrice() );
}
```

Notice the name of this new function. Any function whose name follows the form:

```
operator<C++ operator>
```

is said to *overload* the **specified operator**. When you overload an operator, you're asking the compiler to call your function *instead* of interpreting the operator as it normally would.

56.1.1 Calling an Operator Overloading Function

When the compiler calls an overloading function, it maps the operator's operands to the function's parameters. For example, suppose the function

```
float operator+( MenuItem item1, MenuItem item2 )
{
return( item1.GetPrice() + item2.GetPrice() );
}
```

is used to overload the `+` operator. When the compiler encounters the expression `chicken + wine` it calls `operator+()`, passing `chicken` as the first parameter and `wine` as the second parameter. `operator+()`'s `return` value is used as the result of the expression.

It is important to note that:

- The number of operands taken by an operator determines the number of parameters passed to its overloading function.

For example, a function designed to overload a *unary* operator takes a **single** parameter; a function designed to overload a *binary* operator takes **two** parameters.

56.1.2 Operator Overloading Using a Member Function

You can also use a member function to overload an operator. For example, the function

```
float MenuItem::operator+( MenuItem item )
{
return( GetPrice() + item.GetPrice() );
}
```

overloads the + operator and performs pretty much the same function as the previous example. The difference lies in the way a member function is called by the compiler.

When the compiler calls an overloading member function, it uses the first operand to call the function and passes the remainder of the operands as parameters. So with the function just given in place, the compiler handles the expression

`chicken + wine`

by calling `chicken.operator+()`, passing `wine` as a parameter, as if you had made the following call:

```
chicken.operator+( wine )
```

Again, the value returned by the function is used as the result of the expression.

56.1.3 Multiple Overloading Functions

The previous example brings up an interesting point. What will the compiler do when it encounters *several* functions that overload the same operator?

For example, both of the following functions overload the + operator:

```
float operator+( MenuItem item1, MenuItem item2 )

float MenuItem::operator+( MenuItem item )
```

If both are present, which one is called?

The answer to this question is, **neither**:

- The compiler will not allow you to create an ambiguous overloading situation.
  - You **cannot** overload an operator with similar type operands.
- You can create several functions that overload the same operator, however:
  - You might create one version of operator+() that handles MenuItems and another that allows you to add two arrays together.
  - The compiler chooses the proper overloading function based on the types of the operands.

56.1.4 An Operator Overloading Example

Here's an example that illustartes manu of the above concepts.

We will override the + and *= operators so that can deal with time arithmetic: we'll declare a `Timeclass` and use it to store a length of time specified in hours, minutes, and seconds. Then, we'll overload the + and *= operators and use them to add two times together and to multiply a time by a specified value.

The code listing for `time.cpp` now follows:

```
#include <iostream.h>



//--------------------------------------  Time

class Time
{
// Data members...
private:
short hours;
short minutes;
short seconds;

// Member functions...
void NormalizeTime();
public:
Time();
Time( short h, short m, short s );
void Display();
Time operator+( Time &aTime );
void operator*=( short num );
};

Time::Time()
{
seconds = 0;
minutes = 0;
hours = 0;
```

```
}

Time::Time( short h, short m, short s )
{
seconds = s;
minutes = m;
hours = h;

NormalizeTime();
}

void Time::NormalizeTime()
{
hours += ((minutes + (seconds/60)) / 60);

minutes = (minutes + (seconds/60)) % 60;

seconds %= 60;
}

void Time::Display()
{
cout << "(" << hours << ":" << minutes
<< ":" << seconds << ")\n";
}

Time Time::operator+( Time &aTime )
{
short h;
short m;
short s;

h = hours + aTime.hours;
m = minutes + aTime.minutes;
s = seconds + aTime.seconds;

Time tempTime( h, m, s );
```

```
return tempTime;
}

void Time::operator*=( short num )
{
hours *= num;
minutes *= num;
seconds *= num;

NormalizeTime();
}


//--------------------------------------  main

int main()
{
Time firstTime( 1, 10, 50 );
Time secondTime( 2, 24, 20 );
Time sumTime;

firstTime.Display();
secondTime.Display();

cout << "---------\n";

sumTime = firstTime + secondTime;
sumTime.Display();

cout << "*      2\n";
cout << "---------\n";

sumTime *= 2;
sumTime.Display();

return 0;
}
```

The output of the program is:

```
:
(1:10:50)
(2:24:20)
--------
(3:35:10)
* 2
--------
(7:10:20)
```

First, we declare the `Time` class, which is used to store time in `hours, minutes`, and `seconds`. We also declare the member functions, The first, `NormalizeTime()`, is declared as `private` whilst the rest of the member functions are declared to be `public`.

- The function, `NormalizeTime()`, converts any overflow in the seconds and minutes data members; for example, 70 seconds is converted to 1 minute and 10 seconds. `NormalizeTime()` will only be used from within the Time class. Since we're not planning on deriving any classes from Time, we've left it as private.
- The two constructors: `Time(), Time( short h, short m, short s )`. The first takes no parameters and is used to create a `Time` object with all three data members set to 0 (You'll see why later on in the code). The second `Time` constructor uses its three parameters to initialize the three Time data members and then calls `NormalizeTime()` to resolve any overflow.
- The `Display()` displays the time stored in the current object, `operator+()` overloads the `+` operator, and `operator*=()` overloads the `*=` operator.
- The overload operator `operator+()` is called when the + operator is used to add two `Time` objects together. The first operand is used as the current object, and the second operand corresponds to the parameter `aTime`. Notice that `aTime` is declared as a *reference parameter.*This code would also work if `aTime` were declared without the &. Without the &, the compiler would create a copy of the parameter to pass in to `operator+()`. Since C++ passes its parameters on the stack, this

could cause a problem if the parameter was big enough. With the &, `aTime` is a reference to the object passed in as a parameter.

The function then takes the hours, minutes, and seconds data members of the two objects and adds the together (stored in the local variables `h, m`, and `s`

A new `Time` object `tempTime` is created using `h, m`, and `s`. Finally, we return the newly created object. Since we are not using a reference, the compiler will make a copy of `tempTime`, then return the copy. The compiler is responsible for destroying this copy, so you don't have to worry about it.

- The other overload operator `operator*=()` is called when the `*=` operator is used to multiply a `Time` object by a constant. Notice that `operator*=()` does not return a value because the multiplication is performed *inside* the `Time` object that appears as the first operand. Each of the `Time` object's data members is multiplied by the specified `short num`

  Since we are not creating a new `Time` object, `NormalizeTime()` is called to fix any overflow problems that may have just been caused:

In general, your overloading functions return a value if it makes sense for the operator being overloaded. If the operator includes an =, chances are you'll make your changes in place and won't return a value, as we did with `operator*=()`. If the operator doesn't include an =, you'll most likely return a value, as we did with operator+().

Before you make the decision, build a few expressions using the operator under consideration. Do the expressions resolve to a single value? If so, then you want your overloading function to return a value.

The `main()` function starts by defining two Time objects (the values in parentheses represent the hours, minutes, and seconds, respectively). A third `Time` object, `sumTime`, is created, this time via a call to the `Time` constructor that doesn't take any parameters.

`Display()` is called to display the data members of the two Time objects, and then a line is drawn under the two Times bfore the results of the + and *= operators are displayed

56.2 A Few Overload Restrictions

Therw ae a few restrictions. First, you can only overload C++'s built-in operators with some restrictions (see below) This means that you can't create any new operators. You can't suddenly assign a new meaning to the letter `z`, for example.

You can overload these operators.

```
+ - * / %
^ & | ~ !
, = < > <= >=
== != && ||
++ -- += -= *= /=
%= ^= "&= |=
<<= >>=
[] () -> ->*
new delete
\end{vebatim}




However,  you {\bf cannot} overload
these operators.

\begin{verbatim}
. .* :: ?: sizeof()
```

Note that you **can't change** the way an operator works with a predefined type. For example, you can't write your own `operator()` function to add two `int`s together.

The following rule of thumb for is worth remembering.

If you want the compiler to even consider calling your overloading function, either

- make the function a class member function, *or else*
- make one of its parameters an object.

Remember, the compiler will complain if you write an `operator()` function designed to work solely with C++'s built-in types.

Also note: when you overload the `++` and `--` operators, you'll have to provide two versions of the operator function,

- one to support *prefix* notation, and
- one to support *postfix* notation.

The compiler distinguishes between the two by checking for a dummy int parameter. Prefix version of your operator function shouldn't have any parameters, while the postfix version takes a single `int`.

Here's an example of a prefix and postfix `++` overloading operator for the Time class from our last program.

First, the prefix operator function:

```
Time operator++()
{
*this = *this +1;
return *this;
}
```

Now here's a version of the postfix operator function:

```
Time operator++( int )
{
Time aTime = *this;
*this = *this + 1;
return aTime;
}
```

Notice the unused int parameter in the postfix `operator++()` function. That's how the compiler identifies this function as postfix.

You also **cannot change** an operator's precedence by overloading it. If you want to force an expression to be evaluated in a specific order, you must use *parentheses.*

Overloading functions cannot specify default parameters. This restriction makes sense since a function with default parameters can be called with a variable number of arguments.

For example, you could call the function

```
MyFunc( short a=0, short b=0, short c=0 )
```

using anywhere from zero to three arguments. If an `operator()` function allowed default parameters, you'd be able to use an operator without any operands! If you did that, how would the compiler know which overloading function to call.

You **cannot change** the number of operands handled by an operator. For example, you couldn't make a binary operator unary.

56.3 Mutiple Overloaded Operations

Earlier in the chapter, we looked at a function that overloaded the + operator and was designed to add the price of two MenuItems together:

```
float operator+( MenuItem item1, MenuItem item2 )
{
return( item1.GetPrice() + item2.GetPrice() );
}
```

When the compiler encountered an expression like

```
chicken + wine
```

where both `chicken` and `wine` were declared as `MenuItems`, it called `operator+()`, which passed the two operands as parameters. The `float` produced by adding both prices together was returned as the result of the expression.

What happens when the compiler evaluates an expression like

```
chicken + wine + dessert
```

This expression seems innocent enough, but look at it from the compiler's viewpoint. First, the subexpression

```
chicken + wine
```

is evaluated, resolving to a `float`. Next, this `float` is combined with `dessert` in the expression:

```
<float> + dessert
```

What does the compiler do with this expression?

We designed an overloading function that handles the + operator when its operands are both `MenuItems`, but we don't have one that handles a `float` as the first operand and a MenuItem as the second operand.

Now take a look at the following expression:

```
chicken + (wine + dessert)
\begin{verbatim}

First, the compiler evaluates the subexpression:

\begin{verbatim}
(wine + dessert)
```

resolving it to a `float`. That leaves us with the expression

```
chicken + <float>
```

Once again, we designed an `operator+()` function that handles + and two `MenuItems`, but we don't have one that handles a MenuItem as the first operand and a float as the second operand.

56.3.1 Overloading an Overloading Function

As you can see, you frequently need more than one version of the same `operator()` function. To accomplish this task, you use a technique introduced previously, *function overloading.*

Just as with any other function, you can overload an `operator()` function by providing more than one version, each with its own unique signature.

Remember, a function's signature is based on its para-meter list and not on its return value.

How Many Versions Are Needed?

Figuring out how many versions of an `operator()` function to provide is actually pretty straightforward. Start by making a list of the number of possible types you want to allow for each of the operator's operands. Don't forget to include the type returned by your `operator()` function.

In the previous example, we wanted `operator+()` to handle a float or a MenuItem as either operand, which yields the possibilities shown below:

```
float  + float
float  + MenuItem
MenuItem + float
MenuItem + MenuItem
```

As pointed out earlier, you can't create an `operator()` function based solely on built-in types. Fortunately, the compiler does a perfectly fine job of adding two `float`s together.

With this first case taken care of by the compiler, we're left to construct the remaining three `operator+()` functions.

Our next example program, `menu.cpp`, uses function overloading to do just that.

56.3.2 `menu.cpp`:An Overloader Overloading Example

The code listing for `menu.cpp` is as follows:

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

const short kMaxNameLength = 40;


//-------------------------------------- MenuItem

class MenuItem
{
private:
float price;
char name[ kMaxNameLength ];

public:
MenuItem( float itemPrice, char *itemName );
float GetPrice();
float operator+( MenuItem item );
float operator+( float subtotal );
};

MenuItem::MenuItem( float itemPrice, char *itemName )
{
price = itemPrice;
strcpy( name, itemName );
}

float MenuItem::GetPrice()
{
return( price );
}

float MenuItem::operator+( MenuItem item )
{
cout << "MenuItem::operator+( MenuItem item )\n";

return( GetPrice() + item.GetPrice() );
}

float MenuItem::operator+( float subtotal )
{
cout << "MenuItem::operator+( float subtotal )\n";

return( GetPrice() + subtotal );
}


float operator+( float subtotal, MenuItem item );
// I added the previous line, cause CodeWarrior reports (correctly)
// that there was no prototype for float operator+().
// Now there IS a prototype! Comment out the line
// if you want to see the warning message -- Dave Mark, 10/20/95


//--------------------------------------  operator+()
```

```
float operator+( float subtotal, MenuItem item )
{
cout << "operator+( float subtotal, MenuItem item )\n";

return( subtotal + item.GetPrice() );
}


//--------------------------------------  main()

int main()
{
MenuItem chicken( 8.99, "Chicken Jalfrezi" );
MenuItem wine( 2.99, "Rioja by the Glass" );
MenuItem dessert( 3.99, "Fresh Mangoes" );
float total;

total = chicken + wine + dessert;

cout << "\nTotal: " << total
<< "\n\n";

total = chicken + (wine + dessert);

cout << "\nTotal: " << total;

return 0;
}
```

The output of this program is:

```
MenuItem::operator+( MenuItem item )
operator+( float subtotal, MenuItem item )
Total: 15.969999
MenuItem::operator+( MenuItem item )
MenuItem::operator+( float subtotal )
Total: 15.969999
```

`menu.cpp` starts with two include files (<`iostream.h`¿ and <`string.h`>)and a single constant, `const short kMaxNameLength = 40`.

Next, the `MenuItem` class is declared. The `MenuItem` class contains two data members.`price` lists the price of the item while `name` contains the item's name as it might appear on a menu. Notice that both data members are marked as `private`, which shouldn't be a problem since we won't be deriving any new classes from `MenuItem`.

The `MenuItem` class features four member functions. The constructor, `MenuItem()`, initializes the `MenuItem` data members; the `GetPrice()` function returns the value of the `price` data member.

The two `operator+()` functions handle the cases where a `MenuItem` object appears as the first operand to the + operator. If the second operand is also a `MenuItem`, the first of the two functions is called; if the second operand is a `float`, the second function is called:

The `MenuItem()` constructor copies its first parameter into the `price` data member, and then it uses `strcpy()` to copy the second parameter into the name data member.

Note that we have to write three versions of `operator+()` even though only two are declared in the class:

- The first version of `operator+()` handles expressions of the form

  ```
  <MenuItem> + <MenuItem>
  ```

- The second version of operator+() handles expressions of the form

  ```
  <MenuItem> + <float>
  ```

- The third version of `operator+()` is, *by necessity*, not a member function of any class. To understand why this is so, take a look at the expressions this version of `operator+()` is designed to handle:

  ```
  <float> + <MenuItem>
  ```

  As mentioned earlier, the compiler uses the first operand to determine how the overloading `operator()` function is called. If the first parameter is an object, that object is used to call the `operator()` function

> and all other operands are passed to the function as parameters. If the first parameter is not an object, the compiler's list of candidate overloading functions is reduced to the program's nonclass `operator()` functions. Once a matching function is located, the compiler calls it, passing all of the operands as parameters.

`main()` declares three `MenuItem` objects, initializing each with a price and a name. `main()` also declares a `float total` used to hold the result of our Menu addition. Next, the three `MenuItems` are added together, the result stored in `total = chicken + wine + dessert`, and the printed.

When the compiler encounters the expression

```
chicken + wine + dessert
```

it first processes the sub-expression

```
chicken + wine
```

Since we're adding two `MenuItems` together, the compiler calls the first of our three `operator+()` functions, as shown by the first line of output (`MenuItem::operator+( MenuItem item )`).

Next, this subtotal is used to process the remainder of the expression:

```
<subtotal> + dessert
```

Since we're now adding a float to a MenuItem, the compiler calls the third `operator+()` function, as shown by the next line of output (`operator+( float subtotal, MenuItem item )`)

Once the calculations are complete, the total is printed

Then, the three MenuItems are added together *again*, this time with *the addition of parentheses* wrapped around the last two operands:

```
total = chicken + (wine + dessert);
```

These parentheses force the compiler to start by evaluating the sub-expression

```
(wine + dessert)
```

Once again, we're adding two `MenuItems` together, as shown by the next line of output (`MenuItem::operator+( MenuItem item )`).

Next, this subtotal is used to process the remainder of the expression:

```
chicken + <subtotal>
```

Since we're now adding a `MenuItem` to a `float`, the compiler calls the second `operator+()` function, as shown by the following line of output (`MenuItem::operator+( float subtotal )`)

Finally, the total is printed a second time

56.4 Some Special Cases

The remainder of this chapter is dedicated to a few special cases. Specifically, we'll focus on writing operator() functions that overload the `new, delete, (), [], ->`, and `=` operators.

One characteristic shared by each of these operators is that they can only be overloaded by a nonstatic class member function. Basically, this means that you won't be using the non-class `operator()` function strategy from our previous example for any of the operators in this section.

56.4.1 Overloading `new` and `delete`

There are two ways you can overload `new` and `delete`.

- You can create two member functions named operator `new()` and operator `delete()` as part of your class design. You might do this if you wanted to implement your own memory management scheme for a specific class.

- You can overload the global new and delete operators by providing operator `new()` and operator `delete()` functions that are not members of a class. You might do this if you wanted new and delete to always initialize newly allocated memory.

Whatever your reasons for overloading `new` and `delete`, proceed with caution. No matter how you do it, once you overload new and delete, you are taking on a big responsibility, one that can get you in deep trouble if you don't handle things properly.

An operator `new` Example

Here's a small example you can use as the basis for your own `new` and `delete` `operator()` functions.

```
#include <iostream.h>


//--------------------------------------- Blob

class Blob
{
public:
void *operator new( size_t blobSize );
void operator delete( void *blobPtr, size_t blobSize );
};

void *Blob::operator new( size_t blobSize )
{
cout << "new: " << blobSize << " byte(s).\n";

return new char[ blobSize ];
}

void Blob::operator delete( void *blobPtr, size_t blobSize )
{
cout << "delete: " << blobSize << " byte(s).\n";

delete [] blobPtr;
}


//--------------------------------------- main()

int main()
{
Blob *blobPtr;

blobPtr = new Blob;
delete blobPtr;

return 0;
}
```

The output of this program is:

```
new: 2 byte(s).
delete: 2 byte(s).
```

`new.cpp` defines a class named Blob, which doesn't do much, but it does contain overloading functions for `new` and `delete`.

There are lots of details worth noting in the `new` and `delete operator()` functions.

- Notice the space between the words `operator` and `new` and between `operator` and `delete`. *Without* the space, the compiler might think you were creating a function called `operatornew()` —a perfectly legal C++ function name.
- The operator `new()` returns a `void *`. This is required. In general, your version of `new` will return a pointer to the newly allocated object or block of memory. If your memory management scheme calls for relocatable blocks, you might want to return a handle (pointer to a pointer) instead. The choice is yours.
- The operator new function must take at least one parameter of type `size_t`. The value for this parameter is provided automatically by the compiler and specifies the size of the object to be allocated. Any parameters passed to new will follow the `size_t` in the parameter list.
- The operator `delete` function never returns a value and **must** be declared to return a `void`. `delete` always takes at least one parameter, a pointer to the block to be deleted. The second parameter, a `size_t`, is optional. If you provide it, it will be filled with the size, in bytes, of the block pointed to by the first parameter.

  Sometimes the size passed as the second parameter to operator `delete()` isn't quite what you expected. If the pointer being deleted is a pointer to a base class yet the object pointed to belongs to a class derived from the base class, the second parameter to operator `delete()` will be the *size of the base class.*

There is an exception to this rule. If the base class's destructor is virtual, the size parameter will hold the proper value, the size of the object actually being deleted.

`main()` creates a `new Blob` and then `delete`s it. When the `Blob` is created, the overriding `new` is called. When the object is deleted, the overloaded version of delete is called.

56.4.2 Overloading ( )

The next special case is the function that overloads the `()` operator, also known as the function *call* operator. One reason to overload the function call operator is to provide a shorthand notation for accessing an object's critical data members. As mentioned earlier, `()` can only be overloaded by a nonstatic class member function.

Consider the following an example, `call.cpp`:

```
#include <iostream.h>


//---------------------------------------  Item

class Item
{
private:
float price;

public:
Item( float itemPrice );
// Note that the default value for taxRate has been removed. The book
// uses a default value of 0. As it turns out, section 13,5 of
// the ANSI C++ draft states that an operator function cannot have
// default arguments.
//
// In this version, I just removed the default value for taxRate
// and changed the call stimpyDoll() to stimpyDoll( 0 ) to
// achieve the same result. Thanks to Khurram Quereshi for figuring
// this one out!! -- Dave Mark, 10/26/95
float operator()( float taxRate );
};

Item::Item( float itemPrice )
{
price = itemPrice;
}

float Item::operator()( float taxRate )
{
return( ((taxRate * .01) + 1) * price );
}


//--------------------------------------- main()

int main()
{
Item stimpyDoll( 36.99 );

cout << "Price of Stimpy doll: $" << stimpyDoll( 0 );
cout << "\nPrice with 4.5% tax:  $" << stimpyDoll( 4.5 );

return 0;
}
```

The output is as follow:

```
Price of Stimpy doll: $36.990002
Price with 4.5% tax: $38.654552
```

`call.cpp` starts by defining an `Item` class. An `Item` object represents an item for sale at Uncle Ren's Toy-o-rama.

`Item` features a single data member, `price`, and two member functions, the `Item()` constructor and a function designed to overload the `()` call operator

The `operator()()` function may look odd, but the syntax using two pairs of parentheses is correct:

```
float Item::operator()( float taxRate )
```

The first pair of parentheses designates the operator being overloaded; the second pair surrounds any parameters being passed to the function. In this case, one parameter, `taxRate`, is specified. Notice that `taxRate` has a default value of 0. You'll see why in a minute.

The `operator()()` function takes the specified `taxRate` and applies it to the `Item`'s price, returning the `Item`'s total (`((taxRate * .01) + 1) * price )`)

Since the function call operator can only be overloaded by a class member function, the previous reference to price refers to the data member of the object used in combination with the call operator.

`main()` starts by creating an `Item` object. Here's where the call overload comes into play:

```
cout << "Price of Stimpy doll: $" << stimpyDoll();
```

By taking advantage of the default parameter, the function call

```
stimpyDoll()
```

returns `stimpyDoll`'s price. We could have accomplished the same thing by coding

```
stimpyDoll.price
```

or

```
stimpyDoll( 0 )
```

Next, we use the same function to calculate the cost of the doll with 4.5% tax included:

```
cout << "\nPrice with 4.5% tax: $"
<< stimpyDoll( 4.5 );
```

Once again, we take advantage of the overloaded function call operator. This time, we provide a parameter. Notice that the same overloading function is used for two different (though closely related) purposes.

The key to properly overloading the function call operator is to use it to provide access to a key data member. If your object represents a character string, you might overload `()` to provide access to a substring, using a pair of parameters to provide the starting position and length of the substring.

Another strategy uses `()` as an iterator function for accessing data kept in a sequence or list. Each call to `()` bumps a master pointer to the next element in the list and returns the new data element. No question about it, the function call operator is a useful operator to overload.

56.4.3 Overloading []

Another useful operator to overload is `[]`, also known as the *subscript* operator. Although it can be used for other things, `[]` is frequently overloaded to provide range checking for arrays. You'll see how to do this in a moment.

The subscript overloading syntax is similar to that of the function call operator. In the statement

```
myChar = myObject[ 10 ];
```

the `[]` overloading function belonging to the same class as `myObject` is called with a single parameter, 10. The value returned by the function is assigned to the variable `myChar`.

On the flip side of the coin, the $\widehat{[]}$ overloading function must support a `[]` expression on the left side of the assignment state-ment, like so:

```
myObject[ 10 ] = myChar;
```

The next example program, `subscript.cpp`, shows you how to properly overload `[]`:

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

const short kMaxNameLength = 40;
```

```
//--------------------------------------  Name

class Name
{
private:
char nameString[ kMaxNameLength ];
short nameLength;

public:
Name( char *name );
void operator()();
char &operator[]( short index );
};

Name::Name( char *name )
{
strcpy( nameString, name );
nameLength = strlen( name );
}

void Name::operator()()
{
cout << nameString << "\n";
}

char& Name::operator[]( short index )
{
if ( ( index < 0 ) || ( index >= nameLength ) )
{
cout << "index out of bounds!!!\n";
return( nameString[ 0 ] );
}
else
return( nameString[ index ] );
}


//--------------------------------------  main()
```

```
int main()
{
Name pres( "B. J. Clinton" );

pres[ 3 ] = 'X';
pres();

pres[ 25 ] = 'Z';
pres();

return 0;
}
```

The output is:

```
B. X. Clinton
index out of bounds!!!
Z. X. Clinton
```

`subscript.cpp` starts by defining a `Name` class. The `Name` class is fairly simple. It is designed to hold a `NULL`-terminated string containing a person's name as well as a short containing the length of the string. The member functions include a constructor as well as two operator overloading functions. One function overloads `[]` the other overloads `()`:

The constructor copies the provided string to the nameString data member and places the length of the string in the nameLength data member.

The `()` operator overloading function simply prints the character string in `nameString`:

The `[]` operator overloading function takes a single parameter, an index into the character string. Notice the unusual return type. By specifying a char reference as a return type, the function ensures that the `[]` operator can appear on either side of an assignment statement. Essentially, an expression such as

```
myObject[0]
```

is turned into a `char` variable containing the character returned by the `[]` overloading function:

```
char& Name::operator[]( short index )
```

Here's the real advantage to overloading the [] operator. Before you access the specified character, you can first do some bounds checking, making sure the character is actually in the character string! If the specified index is out-of-bounds, we print a message and point to the first character in the string.

main() first, creates a Name object. Next, the fourth character in the string is replaced by the character X. When pres() is called, the modified string is displayed: B. X. Clinton

Then, the character Z is placed well out-of-bounds and the string is displayed again.

The [] overloading function lets you know that the specified index is out-of-bounds and the assignment is performed on the first character of the string instead:

```
index out of bounds!!!
Z. X. Clinton
```

56.4.4 Overloading − >

Next on the special cases list is the -> operator, also known as the member access operator. Like the other operators presented in this section, overloading -> provides a shorthand notation that can save you code and add an elegant twist to your program.

When the compiler encounters the -> operator, it checks the type of the left-hand operand. If the operand is a pointer, -> is evaluated normally. If the operand is an object or object reference, the compiler checks to see whether the object's class provides an -> overloading function.

If no -> overloading function is provided, the compiler reports an error, since the -> operator requires a pointer, not an object. If the -¿ overloading function is present, the left operand is used to call the overloading function. When the overloading function returns, its return value is substituted for the original left operand, and the evaluation process is repeated. When used this way, the -> operator is known as a smart pointer.

If these rules sound confusing, hold on. The next example, smartPtr.cpp, should explain things:

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

const short kMaxNameLength = 40;


//--------------------------------------  Name

class Name
{
private:
char first[ kMaxNameLength ];
char last[ kMaxNameLength ];

public:
Name( char *lastName, char *firstName );
void DisplayName();
};

Name::Name( char *lastName, char *firstName )
{
strcpy( last, lastName );
strcpy( first, firstName );
}

void Name::DisplayName()
{
cout << "Name: " << first << " " << last;
}


//--------------------------------------  Politician

class Politician
{
private:
Name *namePtr;
short age;
public:
Politician( Name *namePtr, short age );
Name *operator->();
};

Politician::Politician( Name *namePtr, short age )
{
this->namePtr = namePtr;
this->age = age;
}

Name *Politician::operator->()
{
return( namePtr );
}


//--------------------------------------  main()

int main()
{
Name myName( "Clinton", "Bill" );
Politician billClinton( &myName, 46 );

billClinton->DisplayName();

return 0;
}
```

The output is, simply:

```
Name: Bill Clinton
```

`smartPtr.cpp` defines two classes. The `Name` class which holds two zero-terminated strings containing a person's first and last names. and the `Politician` class which represents a politician. To keep things simple, the info is limited to the politician's age and a pointer to a Name object containing the politician's name. The `Politician` class also contains a member function designed to overload the `->` operator. The function returns a pointer to the politician's Name object (the fact that it returns a pointer is key):

`main()` embeds a last and first name into a `Name` object and then uses that object to create a new `Politician` object (so far, no big deal)

There are several problems here. First, `billClinton` is an object and not a pointer, yet it is used with the `->` operator. Second, the member function `DisplayName()` is not a member of the `Politician` class. How can it be called directly from a `Politician` object?

Basically, the `->` overloading function is doing its thing as a smart pointer by bridging the gap between a `Politician` object and a `Name` member function. When the compiler encounters the `->` operator, it checks the type of the left operand. Since `billClinton` is not a pointer, the compiler checks for an `->` overloading function in the `Politician` class.

When the overloading function is found, it is called, using `billClinton` as the current object. The function returns a pointer to a `Name` object. The compiler substitutes this return value for the original, yielding

```
namePtr->DisplayName()
```

The compiler again checks the type of the left operand. This time, the operand is a pointer and the `->` operator is evaluated normally. The `namePtr` is used to call the `Name` function `DisplayName()`, resulting in the output of `Name:  Bill Clinton`

As you can see, overloading the `->` operator provides a shortcut that allows you to run a direct line between two different classes. You can take this model one step further by supposing that the `->` overloading function returns a `Name` object rather than a pointer to a `Name` object. The compiler then substitutes the `Name` object in the original expression and reevaluates:

```
myName->DisplayName();
```

Once again, since the left operand is an object and not a pointer, the left operand's class is examined in search of another `->` overloading function. This substitution and call of `->` overloading functions is repeated until a pointer is returned (the end of the chain is reached). Only then is the `->` operator evaluated in its traditional form. You can use this technique to walk along a chain of objects. Each `->` overloading function evaluates some criteria, returning an object if the search should con-tinue or a pointer if the end condition has been met.

This is pretty Mind Blowing stuff.

56.4.5 Overloading =

The last of the special cases is the `operator=()` function.

Why overload the `=` *operator?*

To best understand why, take a look at what happens when you assign one object to another.

Suppose you define a String class, like this:

```
class String
 {
  private:
    char *s;
    short stringLength;

  public:
    String( char *theString );
 };
```

The data member `s` points to a `NULL`-terminated string. The data member stringLength contains the length of the string. The constructor `String()` initializes both data members. Notice that no memory has been allocated for `s`. This is done inside the constructor.

Now suppose you create a pair of Strings, like this:

```
String source( "from" );

String destination( "to" );
```

And then, you assign one of the String objects to the other, like this:

```
destination = source;
```

What happens?

As it turns out, the = operator copies one object to another by a process called memberwise assignment. Basically, this means that each data member within one object is copied, one at a time, to the corresponding data member in the receiving object.

The trouble with memberwise assignment is in the way it deals with allocated memory, such as you'd find with a null-terminated character string. When one `(char *)` is copied to another, the address stored in the `(char *)` is copied, not the data pointed to by the address. Once the statement

```
destination = source;
```

executes, both Strings point to the same `NULL`-terminated string in memory. The *default* = operator isn't smart enough to allocate the appropriate amount of new memory and then use `strcpy()` to make a copy of the string. That's where `operator=()` comes in.

If you want the ability to assign the contents of one object to another, and the objects contain allocated memory, you'll have to write a smart = overloading function that knows how to do it right.

He is an `operator=()` example, `equals.cpp`;

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


//---------------------------------------  String

class String
{
private:
char *s;
short stringLength;

public:
String( char *theString );
~String();
void DisplayAddress();
String &operator=( const String &fromString );
```

```
};

String::String( char *theString )
{
stringLength = strlen( theString );
s = new char[ stringLength + 1 ];

strcpy( s, theString );
}

String::~String()
{
delete [] s;
}

void String::DisplayAddress()
{
// I added an extra line to the DisplayAddress function
// because both sets of address in the program were
// turning out to be the same. I now print out the string
// along with the string address. Now when you run the program,
// you can see that the first time you print captain and doctor,
// they contain different strings, but the second time, they
// contain the same string, even though their addresses
// didn't change.
// Sorry for any confusion -- Dave Mark 10/31//95
cout << "String address: " << (unsigned long)s << "\n";
cout << "       content: " << s << "\n\n";
}

String &String::operator=( const String &fromString )
{
delete [] s;

stringLength = fromString.stringLength;

s = new char[ stringLength + 1 ];
```

```
strcpy( s, fromString.s );

return( *this );
}


//--------------------------------------- main()

int main()
{
String captain( "Picard" );
String doctor( "Crusher" );

captain.DisplayAddress();
doctor.DisplayAddress();

cout << "-----\n";

doctor = captain;

captain.DisplayAddress();
doctor.DisplayAddress();

return 0;
}
```

The output of this program is:

```
String address: 3259462
String address: 3259472
----
String address: 3259462
String address: 3261024
```

`equals.cpp` starts by defining the `String` class described earlier, with a few additions. The constructor still allocates the memory for the specified string, but now several new functions are added:

```
public:
String( char *theString );
~String();
void DisplayAddress();
String &operator=( const String &fromString );
```

The constructor starts by calculating the length of the specified string, storing the result in `stringLength`. Next, `new` is used to allocate the proper amount of memory (the extra byte is for the `NULL` terminator at the end of the string). Finally, `strcpy()` is called to copy the source string to the data member `s`.

If `s` is declared as an array of fixed size, instead of as a dynamic string pointer, memberwise initialization works just fine since the memory for the array is part of the object itself. Since s points to a block of memory outside the object, memberwise initialization passes it by.

The String destructor (`String()`) uses delete to destroy the array of chars pointed to by s by calling `delete [] s`

The member function `DisplayAddress()` provides a shorthand way of displaying the address of the first byte of a string:

```
cout << "String address: " << (unsigned long)s <<
```

The = overloading function, just like `operator[]()`, this function must return an `l-value`. In this case, we return a reference to a `String` object. We also take a `String` reference as a parameter. Since you can only assign an object to another object of the same class, the type of the return value will always agree with the type of the parameter. `const` in the parameter declaration just marks the parameter as *read only.* `operator=()` starts by freeing up the memory occupied by the old string. Next, the new value for the data member stringLength is copied from the source String. After that, `new` is used to allocate a block for the new string, and `strcpy()` is used to copy the source string into `s`.

Since this is a pointer to the current object, `*this`is the object itself. We return `*this` to satisfy our need to return an `l-value`.

`main()` puts everthing to the test. First, two `String` objects are created and initialized. Next, the address of each `String`'s string is displayed, using the overloaded `()` operator. Then, the object `captain` is assigned to the object `doctor`, and the addresses of the two text strings are again displayed

If the = operator is not overloaded, the address of the `captain` string simply copies into the `doctor` object's s data member and both addresses are the same.

To that all this work is necessary: If you comment out the `operator=()` function (every single line, not just the insides) as well as its declaration inside the String class declara tion, and run the program again. Without the `operator=()` function, the `String` destructor would try to delete the same block of memory twice!

Chapter 57

The `iostream`

The iostream's insertion operator (<<) has been used for all of our output and iostream's extraction operator (>>) has been used for all of our input. While these operators serve us well, there's much more to `iostream` than has been demonstrated so far.

There are 4 include files <iostream.h>, <fstream.h>, <iomanip.h¿, and <strstrea.h> associated with the complete C++ `iostream`. So far we have only used the basic <iostream.h> header.

The `iostream` is a powerful extension of the C++ language. As we will see shortly, we can easily customize iostream so that the >> and << operators recognize your own personally designed data structures and classes. The `iostream` can also be used to write to and read from files or even character arrays. .

57.1 The Character-Based Interface

The `iostream`'s basic unit of currency is the character. Before a number is written to a file, it is converted to a series of chars. When a number is read from the console, it is read as a series of chars and then, if necessary, converted to the appropriate numerical form and stored in a variable.

The `iostream` was designed to support a character-based user interface. As characters are typed on the user's keyboard, they appear on the console. When your program has something to say to the user, it uses iostream to send a stream of characters to the console. If you plan to write programs for environments such as a graphical version of Unix (Motif, X-window), the

Macintosh or MS Windows, you'll probably do all your user-interface development using class libraries that come with your development environment. The `iostream` doesn't know a thing about pull-down menus, windows, or even a mouse, but as you'll see, it's more than a library of user-interface routines.

57.1.1 The `iostream` Classes

Even if your user interface isn't character-based, the `iostream` still has a lot to offer. You can use the same mechanisms you'd use to manage your console I/O to manage your program's file I/O. The same methods you'd use to write a stream of characters to a file can be used to write those same characters to an array in memory. What links these disparate techniques is their common ancestry. The `iostream` library is built upon a set of powerful classes. The `iostream` base class is named `ios`. While you might not work directly with an `ios` object, you'll definitely work with `ios`' members as well as with classes derived from ios.

You've already started to work with two classes derived from `ios`. The `istream` class is designed to handle input from the keyboard. `cin` is an `istream` object that C++ automatically creates for you. The `ostream` class is designed to handle output to the console. `cout`, tt cerr, and `clog` are `ostream` objects that are also automatically created for you. As you've already seen, `cout` is used for standard output. `cerr` and `clog` are used in the same way as `cout`. They provide a mechanism for directing error messages.

Usually `cerr` is tied to the console, although some operating systems (*e.g* Unix) allow you to redirect `cerr`, perhaps sending the error output to a file or to another console. `cerr` is unbuffered which means that output sent to `cerr` appears immediately on the `cerr` device. `clog` is a buffered version of `cerr` and is not supported by all C++ development environments. To decide which error output vehicle to use (`clog` or `cerr`), consult your operating system manual.

57.1.2 The `istream` and `ostream` classes

Up to this point, your experience with iostream has centered on the extraction (>>) and insertion (<<) operators. For example, the following code reads in a number, stores it in a variable, and then prints out the value of the number:

```
short myNum;
cout << "Type a number: ";
cin >> myNum;
cout << "Your number was: " << myNum;
```

There are a couple of things worth noting in this example.

- `iostream` input and output *are buffered*. Just as in C, all input and all output are accumulated in buffers until either the buffers are filled or the buffers are flushed. On the input side, the buffer is traditionally flushed when a carriage return is entered. On the output side, the buffer is usually flushed either when input is requested or when the program ends. Later in the chapter, you'll learn how to flush your own buffers (how exciting!).
- >> eats up white space — >> ignores spaces and tabs in the input stream.

  If you're reading in a series of numbers, this works out pretty well. But if you're trying to read in a stream of text, you might want to preserve the white space interspersed throughout your input. Fortunately, `istream` offers some member functions that read white-space.

`get( )`

The `istream` member function `get()` reads a single character from the input stream. `get()` comes in three different flavors.

- The first version of `get()` takes a char reference as a parameter and returns a reference to an istream object:

  ```
  istream &get(char &destination);
  ```

  Since `get()` is an istream member function, you can use cin to call it (after all, `cin` is just an `istream` object):

  ```
  char c;
  cin.get( c );
  ```

This version of `get()` reads a single character from the input stream, writes the char into its `char parameter (c)`, and then returns the input stream reference (`cin`). Since `get()` returns the input stream, it can be used in a sequence, as in the following example:

```
char c;
short myShort;
cout << "Type a char and a short: ";
cin.get( c ) >> myShort;
```

This code grabs the first character from the input stream and stores it in `c`. Next, the input stream is parsed for a `short`, and the `short` is placed in `myShort`.

- The second version of `get()` is declared as follows:

```
istream &get(char *buffer, int length, char
delimiter = '\n');
```

This version of `get()` extracts up to `length - 1` characters and stores them in the memory pointed to by `buffer`. If the `char` delimiter is encountered in the input stream, the `char` is pushed back into the stream and the extraction stops. For example, the code:

```
char buffer[ 10 ];
cin.get( buffer, 10, '*' );
```

starts to read characters from the input stream. If a `*` is encoun tered, the extraction stops, the `*` is pushed back into the stream, and a `NULL` terminator is placed at the end of the string just read into buffer.

If no `*` is encountered, nine characters are read into buffer, and, again, buffer is `NULL`-terminated. Notice that `get()` reads only]
tt n - 1 characters, where `n` is specified as the second parameter; `get()` is smart enough to save one byte for the `NULL` terminator. If the third parameter is left out, this version of `get()` uses `'\n'` as the terminating character. This allows you to use `get()` to extract a full line of characters without overflowing your input buffer. For example, the code:

```
char buffer[ 50 ];
cin.get( buffer, 50 );
```

reads up to 49 characters or one line from the input stream, whichever is shorter. Either way, the string stored in `buffer` gets `NULL`-terminated.

- The third version of get() is declared as follows:

  ```
  int get();
  ```

  This version of `get()` reads a single character from the input stream and returns the character, cast as an int, as in the follow-ing example:

  ```
  int c;
  while ( (c = cin.get()) != 'q' )
  cout << (char) c;
  ```

  This code reads the input stream, one character at a time, until a `q` is read. Each character is echoed to the console as it is read. The third version of `get()` returns an `int` and not a `char` to allow it to return the end-of-file character. Typically, `EOF` has a value of `-1`. By returning an `int`, `get()` allows for 256 possible `char` values as well as for the end-of-file character.

  Although `EOF` isn't particularly useful when reading from the console, we'll use this version of `get()` later to read the contents of a file.

`getline( )`

Another `istream` member function that you might find useful is `getline()`, which is prototypes by:

```
istream &getline(char *buffer, int length, char delimiter = '\n');
```

`getline()` behaves just like the second version of `get()`, but it returns the delimiter character instead of pushing it back into the input stream.

ignore( )

`ignore()` is used to discard characters from the input stream:

```
istream &ignore(int length = 1, int delimiter = EOF);
```

`ignore()` follows the same basic approach as `getline()`. It reads up to length characters from the input stream and discards them. This extraction stops if the specified delimiter is encountered. Notice that each of these parameters has a default value, which allows you to call `ignore()` without parameters. Here's an example:

```
char buffer[ 100 ];
cin.ignore( 3 ).getline( buffer, 100 );
cout << buffer;
```

This code drops the first three characters from the input stream and then reads the remainder of the first line of input into buffer. Next, the string stored in `buffer` is sent to the console. Notice that the value returned by `ignore()` is used to call `getline()`. This is equivalent to the following sequence of code (but shorter):

```
cin.ignore( 3 );
cin.getline( buffer, 100 );
```

`peek( )`

`peek( )` allows you to sneak a peek at the next character in the input stream *without* removing the character from the stream. It is prototyped by:

```
int peek();
```

Just like the third version of `get()`, `peek( )` returns an `int` rather than a `char`. This allows `peek( )` to return the end-of-file character, if appropriate, which makes `peek( )` perfect for peeking at the next byte in a file.

`put( )`

The `ostream` member function `put()` provides an alternative to the << operator for writing data to the output stream:

```
ostream &put(char c);
```

`put()` writes the specified character to the output stream. It then returns a reference to the stream, so `put()` can be used in a sequence. Here's an example:

```
cout.put( 'H' ).put( 'i' ).put( '!' );
```

As you might have guessed, the preceding line of code produces a `Hi!` message:

putback( )

`putback()` puts the specified `char` back into the input stream, making it the next character to be returned by the next input operation:

```
istream &putback(char c);
```

Note that `c` must be the last character extracted from the stream.

Since `putback()` returns an `istream` reference, it can be used in a sequence, similar to the example combining `ignore()` and `getline()` shown earlier.

`seekg( )` **and** `seekp( )`

The `istream` member function `seekg()` gives you random access to an input stream:

```
istream &seekg(streampos p);
```

Call `seekg()` to position a stream's get pointer exactly where you want it.

A second version of `seekg()` allows you to position the get pointer relative to the beginning or end of a stream or relative to the current get position, this is defined by:

```
istream &seekg( streamoff offset, relative_to direction );
```

In this second version of `seekg()`, the second parameter is one of `ios::beg`, `ios::cur`, or `ios::end`.

The ostream member function `seekp()` gives you random access to an output stream:

```
ostream &seekp(streampos p);
```

Just like `seekg()`, `seekp()` allows you to position a stream's put pointer exactly where you want it. `seekp()` also comes in a second flavor:

```
ostream &seekp( streamoff offset, relative_to direction );
```

57.2 Some Useful Utilities

To aid you with your stream input and output operations, C++ provides a set of standard utilities that you may find useful (plain old ANSI C also provides these routines). To use any of the utilities described in this section, you must include the header file <`ctype.h`>.

Each of the thirteen functions takes an `int` as a parameter. The `int` represents an ASCII character. Two of the functions, `tolower()` and `toupper()`, map this character either to its lowercase or its uppercase ASCII equivalent. For example,

The remaining eleven functions return either 1 or 0, depend ing on the nature of the character passed in. The function

`isalpha()` returns 1 if its argument is a character in the range 'a' through 'z' or in the range 'A' through 'Z'.

The function `isdigit()` returns 1 if its argument is a character in the range '0' through '9'.

The function `isalnum()` returns 1 if its argu-ment causes either `isalpha()` or `isdigit()` to return 1.

The function `ispunct()` returns 1 if the character is a punc-tuation character. The punctuation characters are ASCII charac-ters in the ranges 33-47, 58-64, 91-96, and 123-126 (consult your nearest ASCII chart).

The function `isgraph()` returns 1 if its argument causes `isalpha()`, `isdigit()`, or `ispunct()` to return 1.

`islower()` returns 1 if the character is in the range 'a' through 'z'.

`isupper()` returns 1 if the character is in the range 'A' through 'Z'.

`isprint()` returns 1 if the character is a printable ASCII character.

`iscntrl()` returns 1 if the character is a control character.

`isspace()` returns 1 if the character has an ASCII value in the range 9-13 or if it has a value of 32 (space). Finally, isxdigit() returns 1 if the character is a legal hex digit (0-9, a-f, or A-F).

57.3 Reading Data from a File

The `ifstream` constructor comes in several varieties. The most widely used of these takes two parameters:

```
ifstream(const char* name, int mode=ios::in );
```

The first parameter is a `NULL`-terminated string containing the name of a file to be opened. The second describes the mode used to open the file. The legal modes are described in the table below:

Mode	**Description**
`ios::in`	Input allowed
`ios::out`	Output allowed
`ios::ate`	Seek to EOF at open
`ios::app`	Output allowed, append only
`ios::trunc`	Output allowed, discard existing contents
`ios::nocreate`	Open fails if file doesn't exist
`ios::noreplace`	Open fails if file does exist

They are declared as part of the ios class (defined in <`iostream.h`>). The default mode is `ios::in`, which opens the file for reading.

UNIX (and some other operating systems) support a third, optional parameter for `ifstream` (and for `ofstream` as well). The third parameter specifies the *protection level* used to open the file.

Since you'll most likely want to use the default mode of `ios::in` when you open a file for reading, you can leave off the last parameter when you create an ifstream object:

```
ifstream readMe( "My_File" );
```

This definition creates an `ifstream` object named `readMe`. Next, it opens a file named `My_File` for reading, attaching the open file to `readMe`.

`ifstream` objects have data members that track whether a file is attached to the stream and, if so, whether the file is open for reading. If a file is attached and open for reading, a `get` pointer is maintained that marks how far you've read into the file. Normally, the get pointer starts life at the very beginning of the file.

Once your file is opened for reading, you can use all of the `iostream` input functions described earlier to read data from the file. For example, the following code opens a file and then reads a single character from it:

```
char c;
ifstream readMe( "My File" );
readMe.get( c );
```

57.3.1 The `iostream` State Bits

Every stream, whether an `istream` or an `ostream`, has a series of four state bits associated with it:

```
enum io_state
 {
   goodbit=0,
   eofbit=1,
   failbit=2,
   badbit=4
 };
```

`iostream` uses these bits to indicate the relative health of their associated stream. You can poke and prod these bits yourself, but there are four functions that reflect each bit's setting:

- the function `int good();` returns nonzero if the stream used to call it is ready for I/O. Basically, if `good()` returns 1, you can assume that all is right with your stream and expect that your next I/O operation will succeed.
- The function `int eof();` returns 1 if the last I/O operation puts you at end-of-file.

- The function `dint fail();` returns 1 if the last operation fails for some reason. As an ex-ample, an input operation might fail if you try to read a short but encounter a text string instead.
- The function `int bad();` returns 1 if the last operation fails and the stream appears to be corrupted. When bad() returns 1, you're in TROUBLE.

There is also the function `void clear( int newState=0 );` which is used to reset the state bits to the state specified as a parameter. In general, you should call `clear()` without specifying a param-eter. clear()'s default parameter sets the state bits back to the pristine, good setting. If you don't clear the state bits after a failure, you won't be able to continue reading data from the stream.

For the most part, you should focus on the value returned by `good()`. As long as `good()` returns 1, there's no need to check any of the other functions. Once `good()` returns 0, you can find out why by querying the other three state functions.

The usual way to repeatedly read data is to usw a while loop and an iostream function as its conditional expression, for example:

```
ifstream readMe( "My_File" );

...

while ( readMe.get( c ) )
  cout << c;
```

What causes this while loop to exit?

`readMe.get( c )` returns a reference to `readMe`, correct?

Actually, this is where the C++ compiler displays a little sleight of hand. When the compiler detects an iostream I/O function used where an `int` *is expected*, it uses the current value of good() as the return value for the function. The previous while loop exits when `readMe.get( c )` either fails or hits an end-of-file.

We can do better by using `good( )`

The sample program, `stateBits.cpp`, demonstrates the basics of working with the `iostream` state bits and state bit functions. Close the current project by selecting from the menu.

The full listing for `stateBits.cpp` is a follows:

```
#include <iostream.h>

int main()
{
char done = false;
char c;
short number;

while ( ! done )
{
cout << "Type a number: ";
cin >> number;

if ( cin.good() )
{
if ( number == 0 )
{
cout << "Goodbye...";
done = true;
}
else
cout << "Your number is: " << number << "\n\n";
}
else if ( cin.fail() )
{
cin.clear();

cin.get( c );
cout << c << " is not a number...";
cout << "Type 0 to exit\n\n";
}
else if ( cin.bad() )
{
cout << "\nYikes!!! Gotta go...";
done = true;
}
}
return 0;
}
```

The sample output is as follows:

```
Type a number:
Type a number small enough to fit inside a short, like 256:
Type a number: 256
```

stateBits.cp starts with the usual `#include` <`iostream.h`> `#include` (since we won't be doing any file I/O, there's no need to include < fstream.h>):

`stateBits` creates a loop that reads in a number and then prints the number in the console window. If the number entered is 0, the program exits. Things start to get interesting when a letter is entered instead of a number.

Note that `done` acts as a Boolean logic operator. When it is set to `true`, the loop exits. `c` and `number` are used to hold data read from the console.

We enter the main loop, are prompted for a number and then use >> to read the `number` from the console.

If a number appropriate for a short is typed at the prompt, `cin.good()` returns true:

If the number typed is 0, we say goodbye and drop out of the loop; otherwise, we display the number and start all over again

If the input is of the wrong type (*e.g.* a letter or a `float`), or is a number that is too large (99999) or too small (-72999), the input operation fails and `cin.fail()` returns 1:

If a `fail` is detected, the first thing we must do is call `clear()` to reset the state bits — if we don't clear the state bits back to their healthy state and we won't be able to continue reading data from the stream. Once the state bits are reset, we read the character that caused the the stream to choke. Since we're not trying to interpret this character as a number, this read won't fail. Having read in the offending character, we display it, along with an appropriate message on the console

This example implements a pretty simple-minded recovery algorithm. If you typed in something like `xxzzy`, the loop would fail five times since you knock out only a single character with each recovery. You might want to try your hand at a more sophisticated approach. For example, you might use `cin.ignore()` to suck in all the characters up to and including a carriage return. Better yet, you might use `cin.get()` to read in the remainder of the offending characters and then pack-age them in an appropriate error message.

The final possibility lies with a call to `bad()`. Since the bad bit will likely never be set, you'll probably never see this message.

57.4 Writing Data to a File

Earlier, the `ifstream` constructor was used to open a file for reading:

```
ifstream readMe( "My_File" );
```

In the same way, the ofstream constructor can be used to open a file for writing:

```
ofstream writeMe( "My_File" );
```

The `ofstream` constructor takes two parameters, with `ios::out` used as the default mode parameter. Note that you can pass more than one mode flag at a time. To open a file for writing if the file doesn't already exist, try something like this:

```
ofstream writeMe( "My_File", ios::out | ios::nocreate );
```

The rest of the mode flags as the same as for Reading a file above.

There is a way to open a file for both reading and writing. Use the `fstream` class and pass both the `ios::in` and `ios::out` mode flags, like this:

```
fstream inAndOut( "My_File", ios::in | ios::out );
```

The `fstream` class is set up with two file position indicators, one for reading and one for writing. Prottypes and definitions are found in the include file <`fstream.h`>.

Once your file is open, you can close it by calling the `close()` member function, for example:

```
writeMe.close();
```

In general, this call isn't really necessary since the `ifstream` and `ofstream` destructors automatically close the file attached to their associated stream.

You can also create an `ifstream` or `ofstream` without associating it with a file.

Why would you want to do this?

If you planned on opening a series of files, one at a time, you might want to do this by using a single stream, not by declaring one stream for each file. Using a single stream is more economical. Here's an example:

```
ifstream readMe;
readMe.open( "File_1" );
// Read contents - be sure to include error checking!
readMe.close();
readMe.open( "File_2" );
// Read contents - be sure to include error checking!
readMe.close();
// Repeat this as necessary...
```

57.5 read( ), write( ), and Others

There are some `istream` member functions that are particularly useful when dealing with files. The member function `read()` reads a block of size bytes and stores the bytes in the buffer pointed to by data:

```
istream &read(void *data, int size);
```

As you'd expect, if an end-of-file is reached before the requested bytes are read, the fail bit is set.

The member function `size_t istream::gcount()` returns the number of bytes successfully.

The member function `write()` inserts a block of size bytes from the buffer pointed to by data:

```
ostream &write(const void *data, size_t size);
```

The member function `size_t ostream::pcount()` returns the number of bytes inserted by the preceding `write()` call.

57.6 Customizing the `iostream`

There are times when the standard operators and member functions of `iostream` are not adequate. For example, remember the MenuItem class we declared:

```
class MenuItem
 {
   private:
    float price;
    char name[ 40 ];

   public:
    MenuItem::MenuItem( float itemPrice,char *itemName );
    float MenuItem::GetPrice();
};
```

Now suppose you want to display the contents of a `MenuItem` using `iostream`. You can write a `DisplayMenuItem()` member function that takes advantage of `iostream`, but that is somewhat awkward. If you want to display a `MenuItem` in the middle of a `cout` sequence, you have to break the sequence up, sandwiching a call to `DisplayMenuItem()` in the middle:

```
cout << "Today's special is: ";
myItem.DisplayMenuItem();
cout << "...\n";
```

Wouldn't it be nice if `iostream` knew about MenuItems so that you could do something more convenient, like this:

```
cout << "Today's special is: " << myItem << "...\n";
```

Well there is a way to do this: Using the techniques we have learned from operator overloading, you create an `operator<<()` function that knows exactly how you want your MenuItem displayed.

What's more, you can overload the >> operator, providing an `operator>>()` function that knows how to read in a `MenuItem`. The only restriction on both of these cases is that your >> and << overloading functions **must** return the appropriate stream reference so that you can use the >> and << operators in a sequence.

57.6.1 An >> and << Overloading Example

Our next sample program, `overload.cpp`, extends the `ostream` and `istream` classes by adding functions that overload both >> and <<.

The full code listing is:

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

const short kMaxNameLength = 40;


//--------------------------------------- MenuItem

class MenuItem
{
private:
float price;
char name[ kMaxNameLength ];

public:
void SetName( char *itemName );
char *GetName();
void SetPrice( float itemPrice );
float GetPrice();
};


// I added these two prototypes. They should have been here
// in the first place... -- Dave Mark 10/20/95
istream &operator>>( istream &is, MenuItem &item );
ostream  &operator<<( ostream &os, MenuItem &item );


void MenuItem::SetName( char *itemName )
{
strcpy( name, itemName );
}
```

```
char *MenuItem::GetName()
{
return( name );
}

void MenuItem::SetPrice( float itemPrice )
{
price = itemPrice;
}

float MenuItem::GetPrice()
{
return( price );
}


//-------------------------  iostream operators


istream &operator>>( istream &is, MenuItem &item )
{
float itemPrice;
char itemName[ kMaxNameLength ];

is.getline( itemName, kMaxNameLength );
item.SetName( itemName );

is >> itemPrice;
item.SetPrice( itemPrice );

is.ignore( 1, '\n' );

return( is );
}

ostream &operator<<( ostream &os, MenuItem &item )
{
```

```
os << item.GetName() << " ($"
<< item.GetPrice() << ") ";

return( os );
}


//-------------------------------------- main()

int main()
{
ifstream readMe( "Menu Items" );
MenuItem item;

while ( readMe >> item )
cout << item << "\n";

return 0;
}
```

The output of this program is:

```
Spring Rolls ($2.99)
Hot and Sour Soup ($3.99)
Hunan Chicken ($8.99)
General Tso's Shrimp ($9.99)
Spring Surprise ($15.99)
```

`overload.cpp` starts with some familiar `#includes` and a `const` definition (`const short kMaxNameLength = 40`.

The `MenuItem` class is a slightly modified version of the one in previous examples. For one thing, the constructor is left out. Instead of initializing the data members when a `MenuItem` is created, `iostream` is used to read in a series of `MenuItems` from a file and initialize each data member using the newly added `SetName()` and `SetPrice()` member functions:

`SetName()` is used to set the value of the name data member. `SetPrice()` is used to set the value of the price data member

`GetName()` returns a pointer to the name data member. By giving the caller of this public function direct access to name, we're sort of defeating the purpose of marking name as private. A more appropriate approach might be to have `GetName()` return a copy of name.

`GetPrice()` returns the value of the price data member.

The `operator>>()` function is called by the compiler whenever the >> operator is encountered having an `istream` as its left operand and a MenuItem as its right operand. Since all >> sequences are resolved to `istream` references, the left operand is always an `istream` object. To make this a little clearer, imagine an >> sequence with several objects in it:

```
cin >> a >> b;
```

`iostream` starts by evaluating this expression from the left, as if it were written like this:

```
(cin >> a) >> b;
```

Since the >> operator resolves to an `istream` object, the expression `cin >>` a resolves to `cin`, leaving this:

```
cin >> b;
```

The same logic holds true for the << operator:

```
cout << a << b;
```

As the compiler evaluates this expression from left to right, the left operand of the << operator is always an `ostream` object. The point is, whether `istream` or `ostream`, all an `operator()` function needs to do to support sequences is to return the stream reference passed in as the first parameter.

`operator>>()` reads a single MenuItem object from the specified input stream. First, `getline()` is used to read the item's name. Notice that the second parameter to `getline()` is used to limit the number of characters read in, ensuring that `itemName` doesn't exceed its bounds. `SetName()` is used to copy the entered name into the name data member.

Then, >> is used to read the item's price into `itemPrice`, and `SetPrice()` is used to copy `itemPrice` into the price data member.

When the extraction operator reads the price from the input stream, it leaves the carriage return following the number unread.

`ignore()` is used to grab the carriage return, leaving the stream set up to read the next `MenuItem`.

Finally, the stream passed in to the `operator>>()` function is returned, preserving the integrity of the sequence

`operator<<()` is somewhat simpler. It uses << to write the name and price data members.

Once again, the stream passed in as the first parameter is returned.

`main()` declares an ifstream object and ties it to the file named `Menu` Items. This file contains a list of `MenuItems` with the name and price of each item appearing on its own line.

`main()` also declares a `MenuItem` object named item. Notice that no parameters are passed because there's no constructor to do anything with the parameters.

Next, a `while` loop is used to read in all the `MenuItems` that can be read from the input stream (which is, in this case, a file named `Menu` Items). The overloaded version of >> is used to read in a `MenuItem`, and the overloaded version of << is used to display the MenuItem in the console window.

It's important to note that `operator>>()` and `operator<<()` are designed to work with any input and output stream. In this case, the `MenuItems` are read from a file and displayed in the console window. By making a few changes to `main()` — and not changing the two `operator()` functions — you can easily change the program to read from standard input (you'd probably want to add in a prompt or two) and send the output to a file. This is easy with the `iostream`.

57.6.2 Formatting Your Output

In the preceding program, we overloaded the << operator so that we could display a `MenuItem` precisely the way we wanted it to appear. Unfortunately, there's no way to overload the << operator to customize the appearance of *built-in* data types such as `short` or `float`.

Fortunately, iostream provides several mechanisms that allow you to customize your I/O operations.

In general, `iostream` follows some fairly simple rules when it comes to formatting output. If you insert a single `char` in a stream, exactly one character position is used. When some form of integral data is inserted, the insertion is exactly as wide as the number inserted, including space for a sign, if applicable. No padding characters are used.

When a `float` is inserted, room is made for up to six places of precision to the right of the decimal place. Trailing zeros are dropped. If the number is either very large or very small (how big or how small depends on the implementation), *exponential notation is used.* Again, room is made for a sign, if applicable. For example, the number 1.234000 takes up five character positions in the stream since the trailing zeros are dropped: 1.234

When a string is inserted, each character, not including any `NULL` terminator, takes up one character position.

The Formatting Flags

The `ios` class maintains a set of flags that control various formatting features. You can use the `ios` member functions `setf()` and `unsetf()` to turn these formatting features on and off.

Each feature corresponds to a bit in a bit field maintained by the ios class.

Some features are independent, while others are grouped together. For example, the flag `ios::skipws` determines whether white space is skipped during extraction operations. This feature is not linked to any other features, so it may be turned on and off without impacting any of the other formatting flags.

To turn an independent flag on and off, you use the `setf()` and `unsetf()` member functions as follows:

```
cin.setf( ios::skipws ); // Skip whitespace on input
cin.unsetf( ios::skipws ); // Don't skip whitespace
                           // on input
```

Alternatively, you can use the `flag()` member function to retrieve the current flag settings as a group, OR the new flag into the group, and then use `flag()` again to reset the flag settings with the newly modified bit field:

```
int myFlags;
myFlags = cout.flag(); // returns flag bitfield
myFlags |= ios::skipws; // ORs in skipws flag
cout.flag( myFlags ); // resets flags
```

Unless you really need to work at this level, you're better off sticking with `setf()` and `unsetf()`.

Turning independent flags on and off individually is no problem, but things get interesting when flags are grouped. For example, the `radix` flags determine the default base used to represent numbers in output.

The `radix` flags are `dec`, `oct`, and `hex`, repre senting decimal, octal, and hexadecimal formats, respectively. The problem here is that only *one* of these flags should be turned on at a time. If you use `setf()`, you could easily turn all three flags on, producing unpredictable results.

To handle grouped flags, `setf()` makes use of a second, optional parameter that indicates which group a flag belongs to.

For example, the `radix` flags `dec, oct`, and `hex` belong to the group `basefield`. To set the `hex` flag, you make the following call:

```
cout.setf( ios::hex, ios::basefield );
```

This call ensures that when the specified flag is set, the remainder of the fields in the group get unset.

The grouped flags `left`, `right`, and `internal` are part of the `adjustfield` group. They are used in combination with the `width()` member function.

`width()` determines the minimum number of characters used in the **next** (and *only* next) numeric or string output operation. If the left flag is set, the next numeric or string output operation appears left-justified in the currently specified `width()`. The output is padded with the currently specified `fill()` character. You can use `fill()` to change this padding character.

57.6.3 A Formatting Example,`formatter.cpp`

Let us study a simple formatting example in order to illustrate many of above points.

The full code listing for `formatter.cpp` is as follows:

```
#include <iostream.h>


//--------------------------------------- main()

int main()
```

```
{
cout << 202 << '\n';

cout.width( 5 );
cout.fill( 'x' );
cout.setf( ios::left, ios::adjustfield );

cout << 202 << '\n';

cout.width( 10 );
cout.fill( '=' );
cout.setf( ios::internal, ios::adjustfield );

cout << -101 << '\n';

cout.width( 10 );
cout.fill( '*' );
cout.setf( ios::right, ios::adjustfield );

cout << "Hello";

return 0;
}
```

The output of this program is:

```
202
202xx
-======101
*****Hello
```

`formatter.cp` starts with the standard include file.

`main()` starts by displaying the number 202 in the console in standard fashion

```
cout << 202 << '\n';
```

As you'd expect, this code produces the following line of output: 202.

Next, `width()` is used to set the current width to 5, and `fill()` is used to make `x` the padding character:

Remember, `width()` applies only to the very next string or numeric output operation, even if it is part of a sequence. The padding character lasts until the next call of `fill()` or until the program exits.

If your output operation produces more characters than the current width setting, don't worry. All your characters will be printed.

Now, the left flag is set, asking `iostream` to left-justify the output in the field specified by `width()`:

```
cout.setf( ios::left, ios::adjustfield );
cout << 202 << '\n';
```

When the number 202 is printed again, it appears like this:

```
202xx
```

Then, `width()` is altered to 10, `fill()` is changed to `=`, and the `internal` flag is set. The `internal` flag asks `iostream` to place padding in between a number and its sign, if appropriate, so that it fills the `width()` field:

```
cout.width( 10 );
cout.fill( '=' );
cout.setf( ios::internal, ios::adjustfield );
cout << -101 << '\n';
```

Printing the number -101 produces the following line of output:

```
-======101
```

Finally, `width()` is reset to 10 (otherwise, it would have dropped to its default of 0), `fill()` is set to `*`, and the right flag is set to right-justify the output:

```
cout.width( 10 );
cout.fill( '*' );
cout.setf( ios::right, ios::adjustfield );
cout << "Hello";
```

When the string "Hello" is printed, this line of output appears:

```
*****Hello
```

57.6.4 More Flags and Methods

The `showbase` flag is independent. If it is set, octal numbers are displayed with a leading `zero` and `hex` output appears with the two leading characters `0x`. The `showpoint`, `uppercase`, and `showpos` flags are also independent. If `showpoint` is set, trailing zeros in floating-point output are displayed. If `uppercase` is set, `E` rather than `e` is used in scientific notation and `X` rather than `x` is used in displaying hex numbers. If `showpos` is set, positive numbers appear with a leading `+`.

The `scientific` and `fixed` flags belong to the `floatfield` group. If `scientific` is set, scientific notation is used to display floating-point output. If `fixed` is set, standard notation is used. If neither bit is set, the compiler uses its judgment and prints very large or very small numbers using scientific notation and all other numbers using standard notation. To turn off both bits, you pass a zero instead of fixed or scientific:

```
cout.setf( 0, ios::floatfield );
```

Both the `fixed` and `scientific` flags are tied to the `precision()` member function. `precision()` determines the number of digits displayed after the decimal point in floating-point output:

```
cout.precision( 6 ); // The default for precision...
```

Finally, the `unitbuf` and `stdio` flags are related but not grouped. If `unitbuf` is set, the output buffer is flushed after each output operation. `stdio`, which is only for using *C* I/O, flushes `stdout` and `stderr` after every insertion.

57.7 Manipulators

The `iostream` provides a set of special functions known as *manipulators* that allow you to perform specific I/O operations while you're in the middle of an insertion or an extraction. For example, consider this line of code:

```
cout << "Enter a number: " << flush;
```

This code makes use of the `flush` manipulator. When its turn comes along in the output sequence, the flush manipulator flushes the buffer associated with `cout`, forcing the output to appear immediately as opposed to waiting for the buffer to get flushed naturally.

Just as an I/O sequence can appear in different forms, a manipulator can be called in several different ways. Here are two more examples, each of which calls the flush manipulator:

```
cout.flush(); // Call as a stream member function
flush( cout ); // Call with the stream as a parameter
```

Use whichever form fits in with the I/O sequence you are currently building. If you plan on calling any manipulators that take parameters, be sure to include the file ¡iomanip.h¿. In addition, some iostream implementations require you to link with the math library to use certain manipulators. Check your develop-ment environment manual to be sure.

The Manipulators:

`dec()`, `oct()`, and `hex()` turn on the appropriate format flags, thus turning off the rest of the flags in the basefield group.

`endl()` places a carriage return (`'\n'`) in its output stream and then flushes the stream.

`ends()` places a null character in its output stream and then flushes the stream.

`ws()` eats up all the white space in its input stream until it hits either an end-of-file or the first non-white-space character.

None of the above manipulators take any parameters.

The six remaining to be discussed all take a single parameter and require the included file <`iomanip.h`>.

`setbase(int b)` sets the current radix to either `8, 10`, or `16`.
`setfill(int f)` is a manipulator version of the `fill()` member function.
`setprecision(int p)` is the manipulator version of precision().
`setw(int w)` is the manipulator version of `width()`.
`setiosflags(long f)` is the manipulator version of `setf()`.
`resetiosflags(long f)` is the manipulator version of `unsetf()`.
Here are two manipulator examples. The line

```
cout << setbase( 16 ) << 256 << endl;
```

produces this line of output:

```
100
```

And, the line

```
cout << setprecision( 5 ) << 102.12345;
```

produces this line of output:

```
102.12
```

57.8 The `istrstream` and `ostrstream`

We have met the C `stdio` function sprintf(), there is a similar feature in C++.

Recall `sprintf()` allows you to perform all the standard I/O functions normally associated with `printf()` and `fprintf()` on an array of characters.

The `istrstream` and `ostrstream` classes offer all the power of their ancestor classes (`istream` and `ostream` and, ultimately, `ios`) and allow you to write formatted data to a buffer that you create in memory. Here's an example program, `strstream.cpp`:

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

const short kNumberOfLetters = 26;

//--------------------------------------  main()

int main()
{
ostringstream ostr;
short i;

for ( i = 0; i < kNumberOfLetters; i++ )
ostr << (char)('a' + i);

cout << "Number of characters written: "
<< i << '\n';

cout << "Buffer contents: " << ostr.str();

return 0;
}
```

The output of this program is:

```
Number of characters written: 10
Buffer contents: abcdefghi
```

strstream.cp starts with two `#include`s, the standard <`iostream.h`> and the file required for the `istrstream` and `ostrstream` classes, <`strstrea.h`>.

The constant `kBufferSize` is used to define the size of the buffer that makes up the `ostrstream` object:

m̂ain() creates a buffer to hold the stream's characters. The `ostrstream` constructor takes two parameters, a pointer to the buffer and the size of the buffer. The variable `i` is used to keep track of the number of characters written to the ostrstream

Next, a `while` loop uses `ostr` just as it would use `cout`, writing characters to the stream until an end-of-file causes the loop to terminate. iostream generates the end-of-file when the `put()` pointer points beyond the last character in the stream's buffer (just like its `ifstream` counterpart).

When the loop exits, ten characters, from a to j, have been written to the stream's buffer. The number of characters written to the stream is then displayed.

Next, a `NULL` terminator is written on the last byte of the stream's buffer, creating a `NULL`-terminated string in buffer.

Finally, the contents of the stream are printed.

Just as an `ostrstream` object mirrors the behavior of `cout`, you can create a similar example, using an `istrstream` object, that mirrors the behavior of `cin`. The `istrstream` constructor takes the same two parameters as the `ostrstream` constructor.

Together, istrstream and ostrstream give you a powerful set of tools to use when you work with strings in memory.

57.9 Templates

57.9.1 The Need for Templates

When you design a class, you're forced to make some decisions about the data types that make up that class.

For example, if your class contains an array, the class declaration specifies the array's data type. In the following class declaration, an array of shorts is implemented:

```
class Array
 { private:
     short arraySize;
      // Number of array elements
     short *arrayPtr;
      // Pointer to the array
    public:
      Array( short size );
       // Allocate an array
       // of size shorts
      ~Array();
       // Delete the array
 };
```

In this class,

- the constructor allocates an array of arraySize elements, each element of type `short`.
- The destructor deletes the array.
- The data member `arrayPtr` points to the beginning of the array.
- To make the class truly useful, you'd probably want to add a member function that gives access to the elements of the array.

This Array class works just fine as long as an array of shorts meets your needs.

What happens when you decide that an Array of shorts is not what you need?

Perhaps you need to implement an array of `long`s or, even better, an array of your own data types.

One approach you can use is to make a copy of the Array class (member functions and all) and change it slightly to implement an array of the appropriate type.

For example, here's a version of the `Array` class designed to work with an array of longs:

```
class LongArray
 { private:
      short arraySize;
      long *arrayPtr;
   public:
       LongArray( short size );

       ~LongArray( void );
 };
```

There are definitely problems with this approach.

- You are creating a maintenance nightmare by duplicating the source code of one class to act as the basis for a second class.
- Suppose you add a new feature to your Array class.
- Are you going to make the same change to the LongArray class?

57.9.2 Defining Templates

C++ templates allow you to parameterize the data types used by a class or function.

Instead of embedding a specific type in a class declaration, you provide a template that defines the type used by that class.

An example should make this a little clearer.

Here's a templated version of the Array class presented earlier:

```
template <class T>

class Array
   {
        private:
          short arraySize;
          T *arrayPtr;
        public:
          Array( short size );

          ~Array( void );
   };
```

- The keyword `template` tells the compiler that what follows is not your usual, run-of-the-mill `class` declaration.
- Following the keyword template is a pair of angle brackets (<>) that surround the `template`'s template argument list.
- This list consists of a series of comma-separated arguments
- one argument is the minimum

Once your class template is declared, you can use it to create an object. When you declare an object using a class template, you have to specify a template argument list along with the class name.

Here's an example:

```
Array<long> longArray( 20 );
```

The compiler uses the single parameter, `long`, to convert the `Array` template into an actual class declaration.

This declaration is known as a *template instantiation.*

The instantiation is then used to create the `longArray` object.

A Template Argument List Containing More Than One Type

A `template`'s argument list may contain more than one type.

The `class` keyword **must** precede each argument, and an argument name **can not** be repeated.

Here's an example:

```
template <class Able, class Baker>

class MyClass
  {
     public:
       MyClass( Able param );
```

```
      ~MyClass( void );
      Baker MemberFunction(
         Baker param );
   };
```

Here's a sample definition of a MyClass object:

```
MyClass<long, char *> myObject( 250L );
```

Take a look at the template arguments.
The first, long, will be substituted for `Able`.
The second, `char *`, will be substituted for `Baker`.

57.9.3 Function Templates

The template technique can also be applied to functions.
Here's an example of a function template declaration:

```
template <class T, class U>

T MyFunc( T param1,
          U param2 )
  {
     T var1;
     U var2;

     ......

   }
```

The types defined in the template argument list are then used freely throughout the remainder of the function declaration.

If you use a template to define a function, you must also include the same template information in the function's prototype. Here's a prototype for `MyFunc()`:

```
template <class T, class U>
T MyFunc( T param1,
          U param2 );
```

Function Template Instantiation

When you call a function that has been templated, the compiler uses the parameters passed to the function to determine the types of the template arguments.

Here's a simple example:

```
template <class T>

void MyFunc( T param1 );
```

Suppose this function template were called as follows:

```
char *s;

MyFunc( s );
```

The compiler would match the type of the calling parameter `(char *)` with the type of the receiving parameter (`T`).

In this case, an instantiation of the function is created, and the type `char *` is substituted for `T` everywhere it occurs.

Consider this template:

```
template <class T>

void MyFunc( T param1,
             T param2 );
```

This call of `MyFunc()` won't compile:

```
short i;
int j;
MyFunc( i, j );
```

First, the compiler matches the first parameter and determines that `T` is a short.

When the compiler moves on to the second parameter, it finds that `T` should be an `int`.

Even though an `int` and a `short` are *close*, but since they are not an exact match.

Chapter 58

Multiple Inheritance

58.1 What is Multiple Inheritance?

Our next topic is a variation on an earlier theme, class derivation. In the examples presented in earlier, each derived class was based on a]em single base class. That doesn't have to be the case, however.

C++ allows you to derive a class from more than one base class, a technique known as *multiple inheritance.* As its name implies, multiple inheritance means that a class derived from more than one base class inherits the data members and member functions from each of its base classes.

Why would you want to inherit members from more than one class? Check out the derivation chain in Figure 58.1. The ultimate base class, known as the root base class, in this chain is Computer. The two classes `ColourComputer` and `LaptopComputer` are special types of `Computers`, each inheriting the nonprivate members from `Computer` and adding members of their own as well.

Now we can bring multiple inheritance comes into play. The class `ColourLaptop` is derived from both `ColourComputer` and `LaptopComputer` and inherits members from each class.

Multiple inheritance allows you to take advantage of two different classes that work well together. If you want a program that models a colour, laptop computer and you already have a `ColourComputer` class that manages colour information and a `LaptopComputer` class that manages information about laptops, there is no point reinvent the wheels.Think of the `ColourLaptop` class as the best of both worlds — the union of two already designed classes.

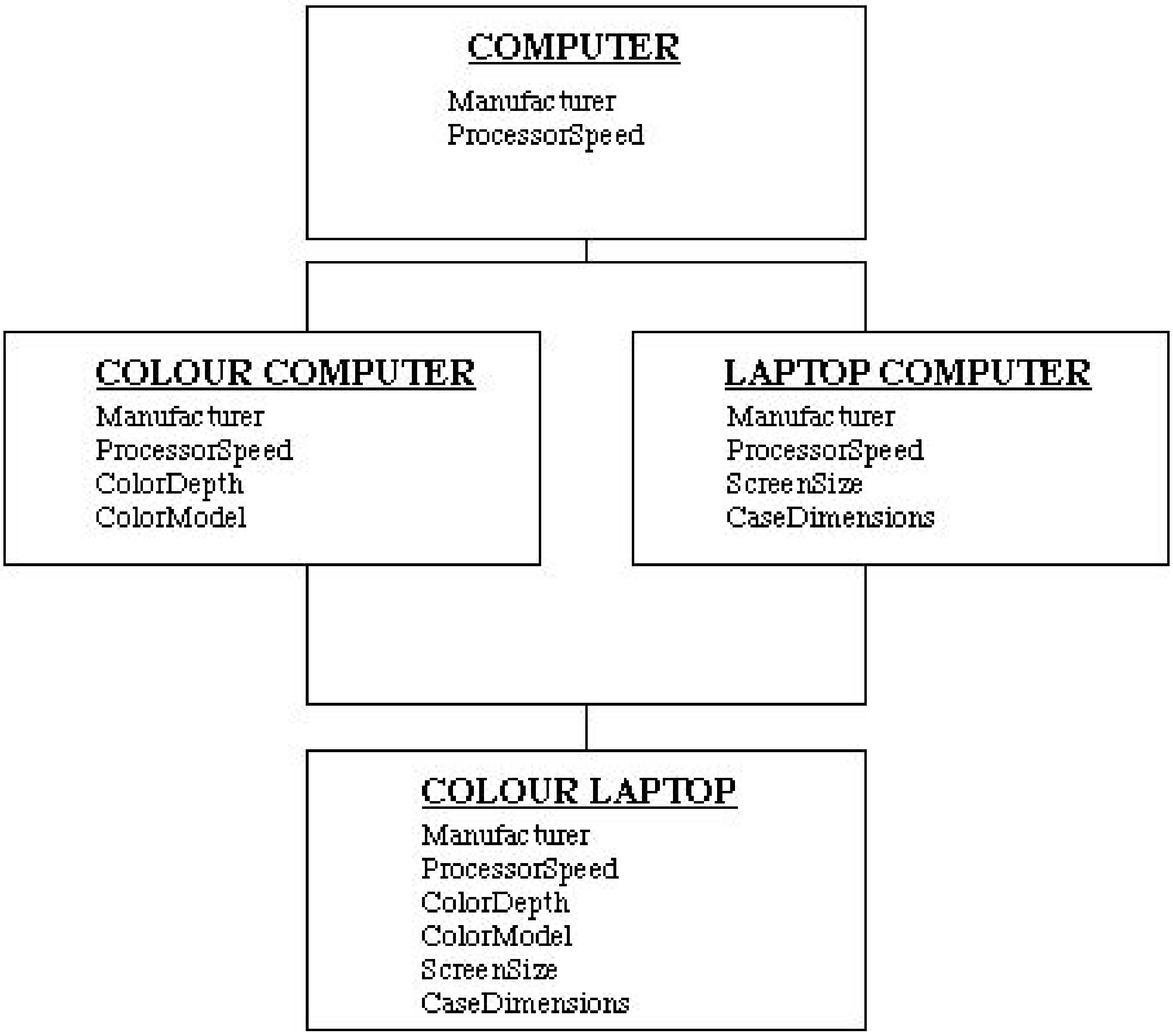


Figure 58.1: Mutliple Inheritance Example

Just as with single inheritance, there are times when multiple inheritance makes sense and times when it is inappropriate. Use the *is a* rule to guide your design. If the derived class *is a* subset of the base class, derivation is appropriate.

In our preceding example, a `ColourComputer` *is a* `Computer` and a `LaptopComputer` *is a* `Computer`. At the same time, a `ColourLaptop` is both a `ColourComputer` and a `LaptopComputer`. This model works just fine.

Let's look at another example.

Imagine a `Date` class and a `Time` class. The `Date` class holds a date, like 07/27/94, while the `Time` class holds a time of day, like 10:24 am. Now suppose you wanted to create a `TimeStamp` class, derived from both the `Date` and `Time` classes.

Would this make sense?

The answer is **NO**: A `TimeStamp` is **not** a `Date` and it is not a `Time`. Instead, a `TimeStamp` *has a* `Date` and *has a* `Time`. When your derivation fits the *has a* model rather than the *is a* model you should rethink your design.

In this case, the `TimeStamp` class *should include* `Date` and `Time` objects as data members, *rather than* using multiple inheritance.

The simple rule is:

- *is a* indicates inheritance.
- *has a* describes the relationship between your derived and base classes, and you should rethink your design.

58.2 A Multiple Inheritance Example, `multInherit.cpp`

Let us now look a complete code eample which demonstrates multiple inheritance as well as a few additional C++ features that you should find interesting.

The full code listing for `multInherit.cpp` is:

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

const short kMaxStringLength = 40;
```

```
//--------------------------------------- Predator

class Predator
{
private:
char favoritePrey[ kMaxStringLength ];

public:
Predator( char *prey );
~Predator();
};

Predator::Predator( char *prey )
{
strcpy( favoritePrey, prey );

cout << "Favorite prey: "
<< prey << "\n";
}

Predator::~Predator()
{
cout << "Predator destructor was called!\n\n";
}


//--------------------------------------- Pet

class Pet
{
private:
char favoriteToy[ kMaxStringLength ];

public:
Pet( char *toy );
~Pet();
};
```

```
Pet::Pet( char *toy )
{
strcpy( favoriteToy, toy );

cout << "Favorite toy: "
<< toy << "\n";
}

Pet::~Pet()
{
cout << "Pet destructor was called!\n";
}


//--------------------------  Cat:Predator,Pet

class Cat : public Predator, public Pet
{
private:
short catID;
static short lastCatID;

public:
Cat( char *prey, char *toy );
~Cat();
};

Cat::Cat( char *prey, char *toy ) :
Predator( prey ), Pet( toy )
{
catID = ++lastCatID;

cout << "catID: " << catID
<< "\n---------\n";
}

Cat::~Cat()
{
```

```
cout << "Cat destructor called: catID = "
<< catID << "...\n";
}

short Cat::lastCatID = 0;


//---------------------------------------  main()

int main()
{
Cat TC( "Mice", "Ball of yarn" );
Cat Benny( "Crickets", "Bottle cap" );
Cat Meow( "Moths", "Spool of thread" );

return 0;
}
```

The output is as follows:

```
Favorite prey: Mice
Favorite toy: Ball of yarn
catID: 1
---------Favorite
prey: Crickets
Favorite toy: Bottle cap
catID: 2
---------Favorite
prey: Moths
Favorite toy: Spool of thread
catID: 3
---------Cat
destructor called: catID = 3...
Pet destructor was called!
Predator destructor was called!
Cat destructor called: catID = 2...
Pet destructor was called!
```

```
Predator destructor was called!
Cat destructor called: catID = 1...
Pet destructor was called!
Predator destructor was called!
```

`multInherit.cpp` starts off with a few `#include`s and a familiar `const short kMaxStringLength = 40`.

Next, three classes are defined.

- The `Predator` class represents a predatory animal
- The `Pet` class represents a housepet.
- The `Cat` class is derived from both the `Predator` class and the `Pet` class – After all, a cat *is a* predator and a cat *is a* pet.

The `Predator` class is pretty simple. It features a single data member, a string containing the predator's favorite prey The `Predator` class also features a constructor and a destructor: The constructor initializes the favoritePrey data member and then prints its value; whilst The destructor prints an appropriate message, just to let you know it was called.

The `Pet` class is almost identical to the `Predator` class, with a favorite toy substituted for a favorite prey.

The `Cat` class is derived from both the `Predator` and `Pet` classes. Notice that the keyword `public` precedes each of the base class names and that the list of base classes is separated by commas:

```
class Cat : public Predator, public Pet
```

`Cat` contains two data members. The first, `catID`, contains a unique ID for each `Cat`. While numbering your cats might not be that useful, if we were talking about `Employees` or `Computers`, a unique employee ID or serial number can be an important part of your class design.

Notice that the second data member, `lastCatID`, is declared using the `static` keyword.

When you declare a data member or member function as *static*, the compiler creates a single version of the member that is shared by all objects in that class.

Why do this?

static members can be very useful. Since a static data member is shared by all objects, you can use it to share information between all objects in a class.

One way to think of a static member is as a global variable whose scope is limited to the class in which it is declared. This is especially true if the `static` member is declared as private or protected.

In this case, `lastCatID` is incremented every time a Cat object is created. Since `lastCatID` is not tied to a specific object, it always holds a unique serial number (which also happens to be the number of `Cat`s created).

The declaration of a `static` data member is just that, a declaration. When you declare a static within a class declaration you need to follow it up with a definition in the same scope.

Typically, you'll follow your class declaration immediately with a definition of the static data member, like this:

```
short Object::lastObjectID;
```

If you like, you can use this definition to initialize the `static` member. `static` data member scope is limited to the file it is declared in.

You'll typically stick your class declaration (along with the class's static member declarations) in a `.h` file.

This is not the case for your static member definition. The definition *should appear* in the `.cpp` file where it will be used.

Along with your `static` data members, you can also declare a static member function. Again, the function is not bound to a particular object and is shared with the entire class.

If the class `MyClass` included a static member function named `MyFunc()`, you could call the function using this syntax:

```
MyClass::MyFunc();
```

Since there is no current object when `MyFunc()` is called, you don't have the advantages of this and any references to other data members or member functions must be done through an object.

`static` member functions are usually written for the sole purpose of providing access to an associated static data member. To enhance your design, you might declare your static data member as private and provide an associated static member function marked as public or protected.

Back to the program.

The `Cat` class has a constructor and a destructor: The Cat constructor maps its input parameters to the `Predator` and `Prey` constructors as follows:

```
Cat::Cat( char *prey, char *toy ) :
Predator( prey ), Pet( toy )
```

The list that follows the constructor's parameter list is called the member initialization list. As you can see, a colon always precedes a constructor's member initialization list.

The Cat destructor also prints a message containing the catID, just to make the program a little easier to follow

Next, the `static` member `lastCatID` is defined. Without this definition, the program wouldn't compile. Notice also that we take advantage of this definition to initialize `lastCatID`.

`static` data members, just like C++ globals, are *automatically* initialized to 0. To make the code a little more obvious, we kept the initialization in there, even though it is redundant.

Finally, `main()` creates three `Cat` objects. Compare the three `Cat` declarations with the program's output. Notice the order of constructor and destructor calls. Note, the destructors are called in the reverse order of the constructors.

58.3 Resolving Ambiguities

Deriving a class from more than one base class brings up an interesting problem. Suppose the two base classes from our previous example, `Predator` and `Pet`, each have a data member with the same name (which is perfectly legal, by the way).

Let's call this data member `clone`. Now suppose that a `Cat` object is created, derived from both `Predator` and `Pet`. When this `Cat` refers to `clone`, which `clone` does it refer to, the one inherited from `Predator` or the one from `Pet`?

As it turns out, the compiler would complain if the `Cat` class referred to just plain `clone` because it can't resolve this ambiguity. To get around this problem, you can access each of the two clones by referring to

```
Predator::clone
```

or

`Pet::clone`

58.4 Multiple Roots

Here's another interesting problem brought on by multiple inheritance. Take a look at the derivation chain in Figure 58.2.

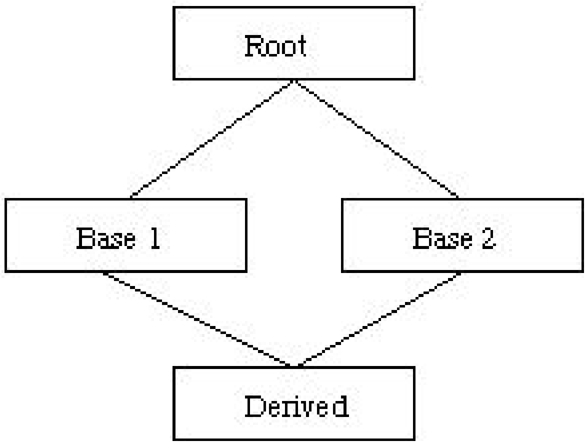


Figure 58.2: Mutliple Root Problem

Notice that the Derived class has two paths of inheritance back to its ultimate base class, Root. Since Derived is derived from both Base1 and Base2, when a Derived object is created, Base1 and Base2 objects are created as well. When the Base1 object is created, a Root object is created. When the Base2 object is created, a second Root object is created.

Why is this a problem?

Suppose Root contains a data member destined to be inherited by Derived. When Derived refers to the Root data member, which of the two Root objects contains the data member Derived is referring to? Sounds like another ambiguity to me.

58.4.1 A Multiple-Root Example, `nonVirtual.cpp`

Before we resolve this latest ambiguity, here's an example that shows what happens when a derived class has two paths back to its root class.

Th code listing for the `nonVirtual.cpp` program is:

```
#include <iostream.h>


//--------------------------------------  Root

class Root
{
public:
Root();
};

Root::Root()
{
cout << "Root constructor called\n";
}


//--------------------------------------  Base1

class Base1 : public Root
{
public:
Base1();
};

Base1::Base1()
{
cout << "Base1 constructor called\n";
}


//--------------------------------------  Base2

class Base2 : public Root
{
public:
```

```
Base2();
};

Base2::Base2()
{
cout << "Base2 constructor called\n";
}


//-------------------------------------- Derived

class Derived : public Base1, public Base2
{
public:
Derived();
};

Derived::Derived()
{
cout << "Derived constructor called\n";
}


//-------------------------------------- main()

int main()
{
Derived myDerived;

return 0;
}
```

The output is as follows:

```
Root constructor called
Base1 constructor called
Root constructor called
```

```
Base2 constructor called
Derived constructor called
```

`nonVirtual.cpp` starts by including <`iostream.h`>:

Four classes are then defined asshown in Figure 58.2.

`Root` consists of a constructor that prints a message letting you know it was called.

`Base1` is derived from `Root`. Its constructor also prints a useful message.

`Base2` is also derived from `Root`. Its constructor also prints a message in the console window.

`Derived` is derived from both `Base1` and `Base2`. Just like all the other classes, `Derived` has its constructor print a message in the console window just to let you know it was called.

`main()` starts the constructor roller coaster by creating a `Derived` object. Since `Base1` is listed first in the `Derived` derivation list, a `Base1` object is created first. Since `Base1` is derived from `Root`, it causes a `Root` object to be created. The `Root` constructor is called and then the `Base1` constructor is called, resulting in the follow ing two lines of output:

```
Root constructor called
Base1 constructor called
```

Next, this process is repeated as a `Base2` object is created. Since `Base2` is also derived from `Root`, it causes a second `Root` object to be created. Once the `Root` constructor is called, control returns to Base2 and its constructor is called:

```
Root constructor called
Base2 constructor called
```

Once the `Base2` object is created, control returns to the `Derived` class and the `Derived` constructor is called: Derived constructor called

58.4.2 The Virtual Base Class Alternative

Once again, think about the problem raised by this last example.

If the `Root` class contained a data member, how would the `Derived` object access the data member?

Which of the two `Root` objects would contain the real copy of the data member?

The answer to this problem lies in the use of *virtual base classes*. We have already declared a member function as virtual to allow a derived class to override the function. Basically, when a virtual function is called by dereferencing a pointer or reference to the base class, the compiler follows the derivation chain down from the root class to the most derived class and looks at each level for a function matching the virtual function. The lowest-level matching function is the one that is called.

Virtual functions are extremely useful. Here's why. Suppose you're writing a program that implements a window-based user interface. Let's say that your standard window is broken into several areas (we'll call them panes) and that each pane is broken into subpanes. When the time comes to draw the contents of your window, your Window class's `Draw()` member function is called. If your `Pane` class also has a `Draw()` member function and if the Window version of `Draw()` is declared as `virtual`, the `Pane`'s `Draw()` is called instead.

This same logic applies to your `SubPane` class and its `Draw()` function. If it is derived from `Pane`, the `SubPane`'s `Draw()` is called instead of the `Pane`'s `Draw()`. This strategy allows you to derive from an existing class using a new class whose actions are more appropriate or more efficient.

A similar technique can be used to remove the ambiguity brought up when a derived class has two different paths back to one of its ancestor classes. In our earlier example, `Root` was the root class, and `Base1` and `Base2` were derived from `Root`.

Finally, `Derived` was derived from both `Base1` and `Base2`. When we created a `Derived` object, we ended up creating two `Root` objects. Thus the ambiguity.

By declaring `Root` as a `virtual` base class, we're asking the compiler to merge the two `Root` object creation requests into a single `Root` object (you'll see how to mark a class as virtual in a moment). The compiler gathers every reference to the `virtual` base class from the different constructor member initialization lists and picks the one that's tied to the deepest constructor. That reference is used, and all the others are discarded. This will become clearer as you walk through the next sample program.

To create a virtual base class, you must insert the `virtual` keyword in the member initialization lists between the virtual base class and the potentially ambiguous derived class. You don't need to mark every class between `Root` and `Derived` as long as the compiler has no path between `Root` and `Derived` that doesn't contain at least one virtual reference. The general strategy is to mark all direct descendants of the virtual base class. In this case, we'd need to place the virtual keyword in both the `Base1` and `Base2` member initialization lists.

Here's an example:

```
class Base1 : public virtual Root
{
public:
Base1();
};
```

The `virtual` keyword can appear either before or after the `public` keyword.

Once the `virtual` keywords are in place, the compiler ignores all member initialization list references to the `Root` class constructor except the deepest one. This sample `Derived` constructor includes a reference to the Root constructor:

```
Derived::Derived( short param ) : Root( param )
{
cout << "Derived constructor called\n";
}
```

Even if the `Base1` and `Base2` constructors map parameters to the `Root` constructor, their mappings are superseded by the deeper, Derived constructor. By overriding the constructor mappings, the compiler makes sure that only a single object of the `virtual` base class (in this case, `Root`) is created.

58.4.3 A Virtual Base Class Example, `virtual.cpp`

This next example brings these techniques to life.

The code listing for `virtual.cpp` is as follows:

```
#include <iostream.h>


//---------------------------------------  Root
```

```
class Root
{
protected:
short num;

public:
Root( short numParam );
};

Root::Root( short numParam )
{
num = numParam;

cout << "Root constructor called\n";
}


//--------------------------------------  Base1

class Base1 : public virtual Root
{
public:
Base1();
};

Base1::Base1() : Root( 1 )
{
cout << "Base1 constructor called\n";
}


//--------------------------------------  Base2

class Base2 : public virtual Root
{
public:
Base2();
```

```
};

Base2::Base2() : Root( 2 )
{
cout << "Base2 constructor called\n";
}


//--------------------------------------  Derived

class Derived : public Base1, public Base2
{
public:
Derived();
short GetNum();
};

Derived::Derived() : Root( 3 )
{
cout << "Derived constructor called\n";
}

short Derived::GetNum()
{
return( num );
}


//--------------------------------------  main()

int main()
{
Derived myDerived;

cout << "-------\n"
<< "num = " << myDerived.GetNum();

return 0;
```

```
}
```

The output of this program is:

```
Root constructor called
Base1 constructor called
Base2 constructor called
Derived constructor called
------
num = 3
```

As usual, `virtual.cpp` starts by including <`iostream.h`>:

This version of the `Root` class includes a data member named `num`. The `Root()` constructor takes a single parameter and uses it to initialize `num` (as you read through the code, try to figure out where the value for this parameter comes from).

`Base1` is derived from `Root`, but it treats `Root` as a

`virtual` base class. Notice that the `Base1()` constructor asks the compiler to call the `Root()` constructor and passes it a value of 1. Will this call take place?

`Base2` also declares `Root` as a `virtual` base class. Now there's no path down from Root that's not marked as `virtual`. The `Base2()` constructor asks the compiler to pass a value of 2 to the `Root()` constructor. Is this the value that is passed on to the Root() constructor?

The `Derived` class doesn't need the virtual keyword (although it wouldn't matter if virtual were used here). The `Derived()` constructor also asks the compiler to pass a value on to the `Root()` constructor. Since `Derived` is the deepest class, this is the constructor mapping that takes precedence. The Root data member num should be initialized with a value of 3. This function makes the value of num available to `main()`. Why can't `main()` reference num directly?

Derived inherits `num` and `main()` doesn't.

`main()` creates a `Derived` object, causing a sequence of constructor calls: Notice that the `Root` constructor is called only once. Finally, the value of `num` is printed.

As you've already seen, `num` has a value of 3, showing that the `Base1` and `Base2` constructor initializations are overridden by the deeper, `Derived` constructor initialization.

Chapter 59

Wrappers

This chapter examines a C++ class that is used as a software layer, or *wrapper* around a utility library written in C.

C++ has many featrure that make it a safer language than C — fewer tedious issues to address.

A C++ wrapper should improve the interface to a library. However, you should take care to ensure that this goal is met.

59.1 Wrapping Up a C libraray

Consider the C stabdard library for intergogating system directories on UNIX. There a five finctions we may wish to use in our C++. We would include them like this:

```
extern "C" {
   DIR *opendir(char *);
   dirent * readdir(DIR *);
   long telldir(DIR *);
   void seekdir( DIR *, long);
   void closedir(DIR *);
};
```

We have already seen these functions and structures in use with C. Here we focus on the C++ wrapping issues.

The `extern "C"` qualifying the function prototypes is a *linkage specification* — indicating that the functions are compiled by a C compiler.

A C++ wrapper class helps to manage the housekeeping by encapsulating pointers to C structures (such as `DIR` and using destructors to ensure that `closedir()` is called for example. However you must make sure you do this yourself.

Having declared the `extern "C"` functions we can now build our Directory wrapper class:

```
class Directory {
  Dir *dir;

  public:
    Directory(char *);
    ~Directory();
    const char *name();
    long tell();
    void seek();
};

Directory::Directory(char *path)
{
    dir = opendir(path);
}

Directory::~Directory()
{
   closedir(dir);
}

const char *Directory::name()

{  dirent *d = readdir(dir);
   return d ?  d->d_name : NULL;
}

long Directory::tell()

{
   return telldir(dir);
}

void Directory::seek(long loc)
{
   seekdir(dir, loc);
}
```

Note that we simply call thye C functions as they are called in C.

59.2 Standard C Header Files

If you look at many of the standard library header files you will see that they are already code up to deal with C++ compilation they have:

```
#ifdef  __cplusplus
extern "C" {
#endif
```

declared near the beginning of the file and

```
#ifdef  __cplusplus
 }
#endif
```

at the end of the file.

If a C++ compiler is being used then the `__cplusplus` macro will be set.

C++ prgrams can therefore safely `#include` standard C header files and have the data in the files safely linked for C++.

Chapter 60

Threads and C++

As we will see in this final example, writing threaded applications is no different in C++ than C.

In fact the best way is to study an example. Earlier when discussing threads (in C) we mentioned an example of multiplying matrices as a good application fro threads. Let's see how we can do this in C++.

60.1 Matrix Multiplication

The matrix multiplication example was written in C++ to show how an object-oriented program might be written to use threads. The concepts and ideas discussed thus far also apply to the C++ language. All the examples in this book could have just as well been written in C++ as in C.

This example simply performs a matrix multiplication, the multiplication of two simple NxN matrices. Essentially, the matrix multiplication performs many mathematical calculations that can be executed independently. This example executes the multiplication of the matrices in parallel by using multiple threads in the processes. The user is allowed to change the size of the matrix objects and the number of threads on which to execute the multiplication.

The example also places all the threaded code into a shared library. This way, all the threaded routines are hidden from the main program. Using this library concept also shows how programs can be changed to use threads without affecting all of the code. In this case, the main program could use a nonthreaded library as well as a threaded one.

This example is rather long, but it demonstrates many of the concepts that have been covered in previous chapters. The program falls into two main parts. The first part is the main thread that creates the matrix objects and starts the matrix multiplication. The second part is the library code, which performs the multiplication on the matrix objects.

The main thread creates the matrix objects, based on user-supplied arguments. The main thread then calls the `MatMult()` routine, which starts the multiplication. A global data structure (thread control block) is used in the library by all the threads in the program. The data structure contains the synchronization variables and other data needed to control the worker threads. This data structure is filled before any threads are created, because the threads use this data during their execution.

The worker threads are created as bound daemon threads. They are bound threads because of the compute nature of the work they do. For all the worker threads to execute in parallel, the level of concurrency would have to be increased, or the threads could be created as bound threads. The threads are also daemon threads because the worker threads should die when the main thread has finished executing. Because this example is compute intensive, the worker threads are created only once; there is no need to recreate the threads for each matrix multiply. The worker threads will always wait for more work to do. If there is no work to be done, then they go to sleep, waiting on a condition variable.

Once there is work to be done, signaled from the main thread, the worker threads wake up and perform the matrix operations on the data specified in the control data structure. At the same time, the main thread waits for all the worker threads to signal that they have finished the work. When the worker threads have finished and have signaled the main thread, they start over again, waiting for more work to do.

The source to Matrix.cpp is:

```
#define _REENTRANT
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <thread.h>
#include "Matrix.h"
```

```
// Main program
main(int argc, char **argv)
{
int size;
int num_threads;
hrtime_t start, stop;

if (argc != 3) {
        cout << "Usage: " << argv[0] << " Matrix-size Threads" << endl;
        exit(0);
        }

// set the size of the matrix and total threads for this run
size = atoi(argv[1]);
num_threads = atoi(argv[2]);
SetMaxThreads(num_threads);

if (size < num_threads) {
        cerr << "The size of the matrix MUST be greater then number of threads."
<< endl;
        exit(1);
        }

cout << "Matrix size: [" << size << "x" << size << "]" << endl;
cout << "Number of worker threads: " << num_threads << endl;

// Create the Matrix
Matrix a('A', size), b('B', size), c('C', size);

// fill A & B with data and clear C
a.fill(); b.fill(); c.clear();

// Start the timer
start = gethrtime();

// Do the matrix multiply
MatMult(a, b, c);
// Stop the timer
stop = gethrtime();

// Print the results -- Only if matrix size is small enough
cout << a << b << c;

// Print the run time
cout << "Matrix multiplication time = "
     << (double)(stop-start)/(double)1000000000
     << " seconds = " << stop-start << " nanoseconds" << endl;
}
```

The source to MatLib.cpp:

```
#define _REENTRANT
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <thread.h>
#include "Matrix.h"

const true = 1;
const false = 0;

// Thread control block - used by all threads as global data
struct thr_cntl_block {
  mutex_t start_mutex;
  cond_t  start_cond;
  mutex_t stop_mutex;
  cond_t  stop_cond;
  Matrix *a, *b, *c;
  int work2do;
  int thrs_running;
  int total_threads;
  int queue;
} TCB;

//////////////////////////////////////////////////////////////////////////
```

```
//  Matrix Class Member Functions
////////////////////////////////////////////////////////////////////////

// Matrix constructor
Matrix::Matrix(char id, int size)
{
matid = id;
matsize = size;
data = new double[matsize*matsize];
}

// Matrix destructor
Matrix::~Matrix()
{
matsize = 0;
matid = 0;
delete[] data;
}

// Fills a matrix object with random data
void Matrix::fill()
{
int i;

for (i=0;i<matsize*matsize;i++)
        data[i] = double(rand()/1000);
srand(rand());
}

// Sets all elements of the matrix to 0.0
void Matrix::clear()
{
int i;

for (i=0;i<matsize*matsize;i++)
        data[i] = .0;
}
```

```
// Prints a Matrix object (if it is small enough)
void Matrix::print(ostream &s) const
{
int i;

if (matsize < 9) {
   s << "Matrix: " << matid << endl;

   for (i=0;i<matsize*matsize;i++)
        {
       s << setiosflags(ios::fixed) << setprecision(1)
             << setw(8) << data[i] << " ";

        if ((i%matsize) == matsize-1) s << endl;
        }

   s << endl << endl;
   }
}

// Overloaded << operator - for ease of printing
ostream &operator<<(ostream &s, const Matrix &mat)
{
   mat.print(s);
   return(s);
}

// Sets the maximum number of threads to use
void SetMaxThreads(int num)
{
TCB.total_threads = num;
}

// The matrix multiply subroutine
MatMult(Matrix &a, Matrix &b, Matrix &c)
{
int static running = false;
int i;
```

```
// Only run this code once, if MatMult is called multiple times
// then there is no need to recreate the threads
if (!running)
    {
    // Initialize the synch stuff.
    mutex_init(&TCB.start_mutex, USYNC_THREAD, 0);
    mutex_init(&TCB.stop_mutex, USYNC_THREAD, 0);
    cond_init(&TCB.start_cond, USYNC_THREAD, 0);
    cond_init(&TCB.stop_cond, USYNC_THREAD, 0);

    // set global variables
    TCB.work2do = 0;
    TCB.thrs_running = 0;
    TCB.queue = 0;
    if (!TCB.total_threads) TCB.total_threads = 1;

    // Create the threads - Bound daemon threads
    for (i = 0; i < TCB.total_threads; i++)
        thr_create(NULL,0, MultWorker, NULL, THR_BOUND|THR_DAEMON, NULL);

    // set the running flag to true so we don't execute this again
    running = true;
    }

// Assign global pointers to the Matrix objects
TCB.a = &a;
TCB.b = &b;
TCB.c = &c;

mutex_lock(&TCB.start_mutex);

  // Assign the number of threads and the amount of work to do
  TCB.work2do = TCB.total_threads;
  TCB.thrs_running = TCB.total_threads;
  TCB.queue = 0;

  // tell all the threads to wake up!
  cond_broadcast(&TCB.start_cond);

mutex_unlock(&TCB.start_mutex);

// yield this LWP
thr_yield();

// Wait for all the threads to finish
mutex_lock(&TCB.stop_mutex);

  while (TCB.thrs_running)
        cond_wait(&TCB.stop_cond, &TCB.stop_mutex);

mutex_unlock(&TCB.stop_mutex);

return(0);
}

// Thread routine called from thr_create() as a Bound Daemon Thread
void *MultWorker(void *arg)
{
  int row, col, j, start, stop, id, size;

  // Do this loop forever - or until all the Non-Daemon threads have exited
  while(true)
    {
      // Wait for some work to do
      mutex_lock(&TCB.start_mutex);

        while (!TCB.work2do)
          cond_wait(&TCB.start_cond, &TCB.start_mutex);

        // decrement the work to be done
        TCB.work2do--;

        // get a unique id for work to be done
        id = TCB.queue++;
      mutex_unlock(&TCB.start_mutex);

      // set up the boundary for matrix operation - based on the unique id
      size = TCB.a->getsize();
      start = id * (int)(size/TCB.total_threads);
      stop = start + (int)(size/TCB.total_threads) - 1;
      if (id == TCB.total_threads - 1) stop = size - 1;

      // print what this thread will work on
      //cout << "Thread " << thr_self() << ": Start Row = " << start
      //     << ", Stop Row = " << stop << endl << flush;
      // Do the matrix multiply - within the bounds set above

      for (row=start; row<=stop; row++)
          for (col = 0; col < size; col++)
              for (j = 0; j < size; j++)
                  TCB.c->getdata()[row*size+col] +=
                  TCB.a->getdata()[row*size+j] *
                  TCB.b->getdata()[j*size+col];

      // signal the main thread that this thread is done with the work
      mutex_lock(&TCB.stop_mutex);
      TCB.thrs_running--;
      cond_signal(&TCB.stop_cond);
      mutex_unlock(&TCB.stop_mutex);
    }
  return 0;
}
```

The source to Matrix.h:

```
#ifndef _matrix_h_
#define _matrix_h_

class Matrix
{
int matsize;
char matid;
double *data;

public:
Matrix(char id, int size);
virtual ~Matrix();
int getsize() {return(matsize);}
double *getdata() {return(data);}
void fill();
void clear();
void print(ostream &s) const;
};

// Function Prototypes
MatMult(Matrix &a, Matrix &b, Matrix &c);       // Matrix Multiply
void *MultWorker(void *arg);                    // Matrix Thread Function
ostream &operator<<(ostream &s, const Matrix &mat);  // Overloaded output
void SetMaxThreads(int num); // Sets the number of threads to use

#endif _matrix_h_
```

This example may look complicated at first, but spend some time here and make sure you understand how this program works. Also, you may want to try running this program with different size matrices and a different number of threads. Here is an example of some test runs, run on a SPARCstation 10 with four 50 MHz superSPARC CPUs:

```
> Matrix 400 1
    Matrix size: [400x400]
    Number of worker threads: 1
    Matrix multiplication time = 44.3368 seconds = 44336817000 nanoseconds
    (This defines the baseline for efficiency.)


> Matrix 400 2
    Matrix size: [400x400]
    Number of worker threads: 2
    Matrix multiplication time = 22.1987 seconds = 22198718000 nanoseconds
    (99.9% efficiency)
```

```
> Matrix 400 3
    Matrix size: [400x400]
    Number of worker threads: 3
    Matrix multiplication time = 14.8932 seconds = 14893245000 nanoseconds
     (99% efficiency)


> Matrix 400 4
    Matrix size: [400x400]
    Number of worker threads: 4
 Matrix multiplication time = 11.6228 seconds = 11622836000 nanoseconds
     (99% efficiency)


> Matrix 400 6
    Matrix size: [400x400]
    Number of worker threads: 6
    Matrix multiplication time = 13.1921 seconds = 13192145000 nanoseconds
     (75% efficiency)
```

Going from one thread to four threads reduced the time needed to perform the matrix multiplication. Also note that running with six threads did not cut the runtime down any more than four threads did, because the workstation had only four CPUs and the extra threads created a scheduling overhead.

Because the multiply routine uses a single global structure, it is not reentrant itself. For CPU-intensive problems such as this, that is not a major problem. Doing one multiply will completely saturate the machine, so there is nothing to be gained from running multiple versions of the multiply routine concurrently.

Chapter 61

Further Reading, Information and References

This chapter gives references to text books used in writing this course and provide further reading on all subjects covered. Information sources on the Internet are also cited where appropriate.

61.1 C References

61.1.1 Basic C and UNIX

The two most appropriate books for the C and standard library aspects of the course are:

Pointers on C, Kenneth Reek, Addison Wesley, 1998.

C Programming in a UNIX Environment, Judy Kay and Bob Kummerfield, Addison Wesley, 1997.

Othere books that are useful:

- Brian W Kernighan and Dennis M Ritchie, The C Programming Language 2nd Ed, Prentice-Hall, 1988.
- A Book on C (4th Ed.), Kelley and Pohl, Addison Wesley, 1998.
- Kenneth E. Martin, C Through UNIX, WCB Group, 1992.
- Keith Tizzard, C for Professional Programmers, Ellis Horwood, 1986.

- Chris Carter, Structured Programming into ANSI C, Pittman, 1991.
- C. Charlton, P. Leng and Janet Little, A Course on C, McGraw Hill, 1992.
- G. Bronson and S. Menconi, A First Book on C: Fundamentals of C Programming (2nd ed.), West Publishing, 1991.

61.1.2 Threads and Remote Procedure Calls

The following books are good resources on Threads:

Pthreads Programming:A POSIX Standard for Better Multiprocessing By Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell, (1st Edition),O'Reilly, 1996

Multithreaded Programming With Pthreads by Bil Lewis, Daniel J. Berg, Prentice Hall Computer Books, 1996

Threads Primer : A Guide to Solaris Multithreaded Programming by Bil Lewis, Daniel J. Berg (Contributor), Bil Bewis, Prentice Hall, 1995.

The following book is a good source of information on Remote Procedure Calls:

Power Programming with RPC, John Bloomer, O'Reilly, 1992.

61.1.3 Internet Resources on C

The web site for this course is

http://www.cm.cf.ac.uk/Dave/C/CE.html

Some good general UNIX/C Web sites can found at

- *http://www.connect.org.uk/techwatch/c/* — Technology Watch : C Archive¡
- *http://www.eecs.nwu.edu/unix.html* — UNIX Reference Desk
- *http://www.bsn.usf.edu:80/ scottb/links/unixprog.html* — Unix Programming Reference.

The Sun Websites has some good information about Threads:

http://www.sun.com/workshop/threads/

61.2 Motif/X Window Programming

61.2.1 Motif/CDE/X Books

There are a number of books that deal with many aspects of the CDE:

- *Common Desktop Environment Advanced User's and System Administrator's Guide*, *Addison-Wesley Developers Press*, 1994.
- *Common Desktop Environment Application Builder User's Guide*, Addison-Wesley Developers Press, 1994.
- *Common Desktop Environment Help System Author's and Programmer's Guide*, Addison-Wesley Developers Press, 1994.
- *Common Desktop Environment Help System Author's and Programmer's Guide*, Addison-Wesley Developers Press, 1994.
- *Common Desktop Environment Programmer's Guide*, Addison-Wesley Developers Press, 1994.
- *Common Desktop Environment Programmer's Overview*, Addison-Wesley Developers Press, 1994.
- *Common Desktop Environment User's Guide*, Addison-Wesley Developers Press, 1994.

There are a number of good general texts on Motif/X Window programming:

- E. Cutler, Gilly D., and T. O'Reilly, *The X Window System in a Nutshell.* O'Reilly & Associates, Sebastopol, CA, USA, 2 edition, 1992.
- F. Culwin, *An X/Motif Programmers Primer.* Prentice Hall, London, UK, 1994.
- *Volume Five: X Toolkit Intrinsics Reference Manual*, O'Reilly & Associates, Sebastopol, CA, USA, 1992.
- D. Heller. *Volume Six A: Motif 1.2 Programming Manual*, O'Reilly & Associates, Sebastopol, CA, USA, 1994.

- D. Heller. *Volume Six B: Motif 1.2 Reference Manual.* O'Reilly & Associates, Sebastopol, CA, USA, 1994.
- E.F. Johnson and Reichard K. *Power Programming: Motif.* O'Reilly & Associates, ew York, USA, 2 edition, 1994.
- L. Mui and E. Pearce. *Volume Eight: X Window System Administrator's Guide.* O'Reilly & Associates, Sebastopol, CA, USA, 1992.
- A. Nye (Ed.). *Volume 0: X Protocol Reference Manual*, O'Reilly & Associates, Sebastopol, CA, USA, 3 edition, 1992.
- A. Nye (Ed.). *Volume Two: Xlib Reference Manual.* O'Reilly & Associates, Sebastopol, CA, USA, 3 edition, 1992.
- J. Newmarch. *The X Window System and Motif: A Fast Track Approach*, Addison Wesley, New York, USA, 1992.
- A. Nye and T. O'Reilly. *Volume Four: X Toolkit Intrinsics Programming Manual (Motif Edition).* O'Reilly & Associates, Sebastopol, CA, USA, 1992.
- A. Nye. *Volume One: Xlib Programming Manual*, O'Reilly & Associates, Sebastopol, CA, USA, 3 edition, 1992.
- Open Software Foundation, London, UK. *OSF/Motif Style Guide*, 1993.
- Open Software Foundation, London, UK. *OSF/Motif 2.0 Programming Manual*, 1995.
- Open Software Foundation, London, UK. *OSf/Motif 2.0 Reference Manual*, 1995.
- Open Software Foundation, London, UK. *OSF/Motif Widget Writer's Guide*, 1995.
- V. Quercia and T. O'Reilly. *Volume Three: X Window System User's Guide.* O'Reilly & Associates, Sebastopol, CA, USA, 1990.
- V. Quercia and T. O'Reilly. *Volume Three: X Window System User's Guide (Motif Edition).* O'Reilly & Associates, Sebastopol, CA, USA, 1991.

- R.K Rost. *X and Motif Quick Reference Guide.* Digital Press, New York, USA, 2 edition, 1993.
- L. Reiss and J. Radin. k *X Window: Inside and Out*, McGraw Hill, New York, USA, 1992.

61.2.2 Motif distribution

Various components of X/Motif are distributed by the OSF, the X Consortium, the Open Group and by a number of independent vendors for a variety of platforms. Section 35.6.1 gives details on these matters.

61.2.3 WWW and Ftp Access

The main WWW source of information for motif is *MW3: Motif on the World Wide Web* (URL: *http://www.cen.com/mw3/*). From the home page you can connect to a wealth of resources for Motif and X Window System development. MW3 presently contains over 700 links. Virtually all aspects of Motif are covered here.

Other useful WWW links include:

- *http://www.rahul.net/kenton/xsites* — Good source of X information and links to many related sites.
- *http://www.landfield.com/faqs/faqsearch.html* — The best frequently asked questions (FAQ) search interface (Usenet Hypertext FAQ Archive).
- X, Xt and Motif FAQ are also archived at:
  - Utrecht University (*http://www.cs.ruu.nl/wais/html/na-dir/*),
  - Oxford University (*http://www.lib.ox.ac.uk/internet/news/faq/*),
  - SUNSite Northern Europe (*http://src.doc.ic.ac.uk/usenet/usenet-by-hierarchy/comp/windows/x/*).

The Motif FAQ is available via ftp also at:

- Century Computing Inc, USA — The file is available in raw text and compressed formats: *ftp://ftp.cen.com/pub/Motif-FAQ,* *ftp://ftp.cen.com/pub/Motif-FAQ.Z* and *ftp://ftp.cen.com/pub/Motif-FAQ.gz.*

- MIT — The Motif FAQ is available in 9 parts: *ftp:// rtfm.mit.edu/pub/usenet-by-group/comp.windows.x.motif.*
- X Consortium — *ftp://ftp.x.org/contrib/faqs/Motif-FAQ*.

61.2.4 Valuable Information Resources

Other sources of information on the Internet are provided via *mailing lists* and *news groups.* Mailing lists are sent via email and serve as discussion groups and avenues for news announcements for particular topics. News groups can be read by specific *news reader* applications and broadly serve as discussion groups. Mailing lists and news groups do not necesarily require a WWW browser for reading although browsers such as Netscape Navigator do provide specific access to news groups and email.

Mailing lists

The following public mailing lists are maintained by the X Consortium for the discussion of issues relating to the X Window System. All are hosted `@x.org`.

xpert A mailing list that discuses many X related issues. This list is gatewayed to the newsgroup comp.windows.x (*see below*).

To subscribe to this mailing list, send mail to the request address. In general this is specified by adding -request to the name of the desired list. Thus, to *add* yourself to the *xpert* mailing list:

```
To: xpert-request@x.org
Subject: (none needed)

subscribe
```

To *unsubscribe*:

```
To: xpert-request@x.org
Subject: (none needed)

unsubscribe
```

To add an address to a specific list, or to add a specific user, you can specify options to the subscribe or unsubscribe command. This example adds dave@widget.uk to the xpert mailing list:

```
To: xpert-request@x.org
Subject: (none needed)

subscribe xpert dave@widget.uk
```

xannounce This is a moderated mailing list for announcing releases of non-commercial X related software, and other issues concerning the X Window System.

This mailing list is gatewayed to the newsgroup *comp.windows.x.announce*.

Subscription requests should be sent to *xannounce-request@x.org*.

News groups

The news group **comp.windows.x.motif** is the main news group for Motif related issues. The following news groups exist for the discussion of other issues related to the X Window System:

comp.windows.x — This news group is gatewayed to the xpert mailing list (*see above*).

comp.windows.x.announce This group is moderated by the staff of X Consortium, Inc. Traffic is limited to major X announcements, such as X Consortium standards, new releases, patch releases, toolkit releases and X conferences, exhibitions, or meetings.

comp.windows.x.apps — This news group is concerned with X applications.

comp.windows.x.intrinsics — This news group is concerned with Xt toolkit.

comp.windows.x.pex — This news group is concerned with the 3D graphics extension to X.

alt.windows.cde — The news group dedicated to Common Desktop Environment issues.

Most of the above news groups have a frequently asked question section posted regularly to the news group which provide valuable information for the novice and discuss common problems. The *comp.windows.x.motif* are also accessible from many of the WWW sites listed in Section 61.2.3

61.3 C++

There are many books on C++, Here are a few good general programming guides:

- *The C++ Programming Language*, B. Stroustrup, Addison Wesley, 1997.
- *Learning C++*, Neill Graham, Mcgraw Hill, 1991.
- *Learning C++: A Hands on Approach*, E. Nagler, West Publishing, 1993.
- *Introduction to C++*, D. Dench and B. Proir, Chapmann Hall, 1994.

Some books which are more advances and deal with Object Oriented Design with C++:

- *C++ Programming Style*, T. Cargill, Addison Wesley, 1992.
- *From Chaos to Classes: Object Oriented Software Development in C++*, D. Duffy, McGraw-Hill, 1995.
- *Object Oriented Software in C++*, M. Smith, Chapmann Hall, 1993.

Thhe major C++ Web repository is:
http://www.austinlinks.com/CPlusPlus/

Appendix A

C Compiler Options and the GNU C++ compiler

This appendix gives some common compiler options and some details on the GNU C/C++ compiler.

A.1 Common Compiler Options

Here we list common C Compiler options. They can be tagged on to the compiler directive. Some take an additional argument.

E.g.

```
cc -c -o prog prog.c
```

The `-o` option needs an argument, `-c` does not.

```
-c        Suppress linking with ld(1) and produce a .o  file
          for each source file.  A single object file can be
          named explicitly using the -o option.

-C        Prevent the C preprocessor from  removing
          comments.

-D        Define symbols either as identifiers (-Didentifer) or as values
          (-Dsymbol=value}) in a similar fashion as the #define preprocessor
          command.
```

```
-E        Run the source file through  the  C  preprocessor,
          only.  Sends the output to the standard output, or
          to a file named with the -o option.  Includes  the
          cpp line numbering information.  (See also, the -P
          option.)

-g        Produce additional symbol  table  information  for
          dbx(1) and dbxtool(1).  When this option is given,
          the -O and -R options are suppressed.

-help     Display helpful information about compiler.

-Ipathname
          Add pathname to the list of directories  in  which
          to   search   for  #include  files  with  relative
          filenames  (not  beginning  with  slash  /).   The
          preprocessor  first searches for #include files in
          the  directory  containing  sourcefile,  then   in
          directories  named  with  -I options (if any), and
          finally, in /usr/include.

-llibrary Link with  object  library  library  (for  ld(1)).
          This option must follow the sourcefile arguments.

-Ldirectory
          Add directory to the list of directories  contain-
          ing  object-library  routines  (for  linking using
          ld(1).

-M        Run only the macro preprocessor  on  the  named  C
          programs,  requesting  that  it  generate makefile
          dependencies and send the result to  the  standard
          output  (see  make(1)  for details about makefiles
          and dependencies).


-o outputfile
```

```
             Name the output file outputfile.  outputfile  must
             have  the  appropriate suffix for the type of file
             to be produced  by  the  compilation  (see  FILES,
             below).  outputfile  cannot be the same as source-
             file (the compiler will not overwrite  the  source
             file).

   -O[level] Optimize the object code.  Ignored when either  -g
             or  -a  is  used.   -O  with  the level omitted is
             equivalent to -O2.  level is one of:

                  1    Do postpass assembly-level  optimization
                       only.

                  2    Do global  optimization  prior  to  code
                       generation,   including  loop  optimiza-
                       tions, common subexpression elimination,
                       copy propagation, and automatic register
                       allocation. -O2 does not optimize refer-
                       ences  to  or definitions of external or
                       indirect variables.


             If the optimizer runs out of memory, it  tries  to
             recover  by  retrying  the  current procedure at a
             lower level of optimization and resumes subsequent
             procedures at the original level.

    -P        Run the source file through  the  C  preprocessor,
             only.  Puts the output in a file with a .i suffix.
             Does not include cpp-type line number  information
             in the output
```

A.2 GCC - The GNU C/C++ Compiler

A.2.1 Introduction to GCC

Information in the section was taken from the GNU CC web site and GNU CC supporting documentation distrubuted with the GNU CC compiler. For more complete and up to date information readers are urgent to consult these sources.

The GNU CC web site URL is: *http://www.gnu.ai.mit.edu/software/gcc/gcc.html*

GCC was developed by GNU to provide a free compiler for the GNU system. GCC is distributed under the terms of the GNU General Public License(20k characters) (GNU GPL for short).

GCC can compile programs written in C, C++, Objective C, Ada 95, Fortran 77, and Pascal (see compiling other languages).

GCC is a full featured compiler, providing everything you need in a C compiler. GCC is updated to support new features and new platforms. The GNU project also provides many companion tools, such as GNU make and GDB (GNU Debugger).

GCC is short for the *GNU C Compiler*; we also sometimes use the name *GNU CC*.

'gcc' is also the command name used to invoke the compiler. The reason for this name is that the compiler initially supported only the C language. Now, 'gcc' will invoke the proper compiler files for C++ files if their names end in '.C', '.cc', '.cpp', or '.cxx'. The 'gcc' command also recognizes Objective C, Pascal, Fortran and Ada files based on their file names.

A.2.2 Languages compiled by GCC

The main GCC distribution includes the source for the C, C++, and Objective C front ends, giving gcc the ability to compile programs in these languages. Additional front ends for Ada 95, Fortran 77, and Pascal are distributed separately. (Note: The front end for Ada is not distributed by the Free Software Foundation, because it is written in Ada and therefore has to be distributed together with binaries for bootstrapping.)

G++ is the C++ compiler. G++ is a compiler, not merely a preprocessor; G++ builds object code directly from your C++ program source. There is no intermediate C version of the program.

Avoiding an intermediate C representation of the program means that you get better object code, and better debugging information. The GNU debugger, GDB, works with this information in the object code to give you comprehensive C++ source-level editing capabilities.

A.2.3 Portability and Optimization

GCC is a fairly portable optimizing compiler which performs many optimizations.

Portability :

GCC supports full ANSI C, traditional C, and GNU C extensions (including: nested functions support, nonlocal gotos, and taking the address of a label).

GCC can generate a.out, COFF, ELF, and OSF-Rose files when used with a suitable assembler. It can produce debugging information in these formats: BSD stabs, COFF, ECOFF, ECOFF with stabs, and DWARF and DWARF 2. Position-independent code is generated for the Clipper, Hitachi H8/300, HP-PA (1.0 & 1.1), i386/i486/Pentium, m68k, m88k, SPARC, and SPARClite.

GCC can open-code most arithmetic on 64-bit values (type 'long long int'). It supports extended floating point (type 'long double') on the 68k and ix86; other machines will follow. GCC generates code for many CPUs. Using the configuration scheme for GCC, building a cross-compiler is as easy as building a native compiler.

Optimizations :

Automatic register allocation Common sub-expression elimination (CSE) (including a certain amount of CSE between basic blocks). Invariant code motion from loops Induction variable optimizations Constant propagation and copy propagation Delayed popping of function call arguments Tail recursion elimination Integration of in-line functions and frame pointer elimination Instruction scheduling Loop unrolling Filling of delay slots Leaf function optimization Optimized multiplication by constants The ability to assign attributes to instructions Many local optimizations automatically deduced from the machine description

A.2.4 GNU CC Distribution Policy

GCC is distributed under the terms of the GNU General Public License (GPL).

Under the GNU GPL, any modified version of GCC, any program that contains any of the code of GCC, must be released as free software–to do otherwise is copyright infringement.

It is permissible to compile non-free programs with GCC. Compiling a program with GCC and distributing the binary does not require you to make the program free software or release its source code. This is because the run-time library included with GCC comes with special permission to link it with your compiled programs without restriction. The legal rules for using the output from GCC are the determined by the program that you are compiling, not by GCC.

However, making programs free software is the right thing to do on general ethical principles, regardless of what compiler you use.

A.2.5 Compile C, C++, or Objective C

The C, C++, and Objective C versions of the compiler are integrated; the GNU C compiler can compile programs written in C, C++, or Objective C.

"GCC" is a common shorthand term for the GNU C compiler. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs.

When referring to C++ compilation, it is usual to call the compiler "G++". Since there is only one compiler, it is also accurate to call it "GCC" no matter what the language context; however, the term "G++" is more useful when the emphasis is on compiling C++ programs.

We use the name "GNU CC" to refer to the compilation system as a whole, and more specifically to the language-independent part of the compiler. For example, we refer to the optimization options as affecting the behavior of "GNU CC" or sometimes just "the compiler".

Front ends for other languages, such as Ada 9X, Fortran, Modula-3, and Pascal, are under development. These front-ends, like that for C++, are built in subdirectories of GNU CC and link to it. The result is an integrated compiler that can compile programs written in C, C++, Objective C, or any of the languages for which you have installed front ends.

In this manual, we only discuss the options for the C, Objective-C, and

C++ compilers and those of the GNU CC core. Consult the documentation of the other front ends for the options to use when compiling programs written in other languages.

G++ is a compiler, not merely a preprocessor. G++ builds object code directly from your C++ program source. There is no intermediate C version of the program. (By contrast, for example, some other implementations use a program that generates a C program from your C++ source.) Avoiding an intermediate C representation of the program means that you get better object code, and better debugging information. The GNU debugger, GDB, works with this information in the object code to give you comprehensive C++ source-level editing capabilities.

A.2.6 GNU CC Command Options

When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the '-c' option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Most of the command line options that you can use with GNU CC are useful for C programs; when an option is only useful with another language (usually C++), the explanation says so explicitly. If the description for a particular option does not mention a source language, you can use that option with all supported languages.

See section Compiling C++ Programs, for a summary of special options for compiling C++ programs.

The gcc program accepts options and file names as operands. Many options have multiletter names; therefore multiple single-letter options may not be grouped: '-dr' is very different from '-d -r'.

You can mix options and other arguments. For the most part, the order you use doesn't matter. Order does matter when you use several options of the same kind; for example, if you specify '-L' more than once, the directories are searched in the order specified.

Many options have long names starting with '-f' or with '-W'—for example, '-fforce-mem', '-fstrength-reduce', '-Wformat' and so on. Most of these have both positive and negative forms; the negative form of '-ffoo' would be '-fno-foo'. This manual documents only one of these two forms, whichever one is not the default.

Here is a summary of all the options, grouped by type. Explanations are in the following sections.

```
Overall Options
       See section Options Controlling the Kind of Output.

       -c  -S  -E  -o file  -pipe  -v  -x language

C Language Options
       See section Options Controlling C Dialect.

       -ansi  -fallow-single-precision -fcond-mismatch  -fno-asm
       -fno-builtin  -fsigned-bitfields  -fsigned-char
       -funsigned-bitfields  -funsigned-char  -fwritable-strings
       -traditional  -traditional-cpp  -trigraphs

C++ Language Options
       See section Options Controlling C++ Dialect.

       -fall-virtual  -fdollars-in-identifiers  -felide-constructors
       -fenum-int-equiv -fexternal-templates  -fhandle-signatures
       -fmemoize-lookups  -fno-default-inline -fno-gnu-keywords
       -fnonnull-objects  -foperator-names  -fstrict-prototype
       -fthis-is-variable -nostdinc++ -traditional  +en

Warning Options
       See section Options to Request or Suppress Warnings.

       -fsyntax-only  -pedantic  -pedantic-errors
       -w  -W  -Wall  -Waggregate-return  -Wbad-function-cast
       -Wcast-align -Wcast-qual  -Wchar-subscript  -Wcomment
       -Wconversion -Wenum-clash  -Werror  -Wformat
       -Wid-clash-len  -Wimplicit  -Wimport  -Winline
       -Wlarger-than-len  -Wmissing-declarations
       -Wmissing-prototypes  -Wnested-externs
       -Wno-import  -Woverloaded-virtual -Wparentheses
       -Wpointer-arith  -Wredundant-decls -Wreorder -Wreturn-type -Wshadow
       -Wstrict-prototypes  -Wswitch  -Wsynth  -Wtemplate-debugging
       -Wtraditional  -Wtrigraphs -Wuninitialized  -Wunused
       -Wwrite-strings

Debugging Options
       See section Options for Debugging Your Program or GNU CC.

       -a  -dletters  -fpretend-float
       -g  -glevel -gcoff  -gdwarf  -gdwarf+
       -ggdb  -gstabs  -gstabs+  -gxcoff  -gxcoff+
       -p  -pg  -print-file-name=library  -print-libgcc-file-name
       -print-prog-name=program  -print-search-dirs  -save-temps

Optimization Options
       See section Options That Control Optimization.

       -fcaller-saves  -fcse-follow-jumps  -fcse-skip-blocks
       -fdelayed-branch   -fexpensive-optimizations
       -ffast-math  -ffloat-store  -fforce-addr  -fforce-mem
       -finline-functions  -fkeep-inline-functions
       -fno-default-inline  -fno-defer-pop  -fno-function-cse
       -fno-inline  -fno-peephole  -fomit-frame-pointer
       -frerun-cse-after-loop  -fschedule-insns
       -fschedule-insns2  -fstrength-reduce  -fthread-jumps
       -funroll-all-loops  -funroll-loops
       -O  -O0  -O1  -O2  -O3

Preprocessor Options
       See section Options Controlling the Preprocessor.

       -Aquestion(answer)  -C  -dD  -dM  -dN
       -Dmacro[=defn]  -E  -H
       -idirafter dir
       -include file  -imacros file
       -iprefix file  -iwithprefix dir
       -iwithprefixbefore dir  -isystem dir
       -M  -MD  -MM  -MMD  -MG  -nostdinc  -P  -trigraphs
       -undef  -Umacro  -Wp,option

Assembler Option
       See section Passing Options to the Assembler.

       -Wa,option

Linker Options
       See section Options for Linking.

       object-file-name  -llibrary
       -nostartfiles  -nodefaultlibs  -nostdlib
       -s  -static  -shared  -symbolic
       -Wl,option  -Xlinker option
       -u symbol

Directory Options
       See section Options for Directory Search.

       -Bprefix  -Idir  -I-  -Ldir

Target Options
       See section Specifying Target Machine and Compiler Version.

       -b machine  -V version

Machine Dependent Options
       See section Hardware Models and Configurations.

       M680x0 Options
       -m68000  -m68020  -m68020-40  -m68030  -m68040  -m68881
       -mbitfield  -mc68000  -mc68020  -mfpa  -mnobitfield
       -mrtd  -mshort  -msoft-float

       VAX Options
       -mg  -mgnu  -munix
```

```
SPARC Options
-mapp-regs  -mcypress  -mepilogue  -mflat  -mfpu  -mhard-float
-mhard-quad-float  -mno-app-regs  -mno-flat  -mno-fpu
-mno-epilogue  -mno-unaligned-doubles
-msoft-float  -msoft-quad-float
-msparclite  -msupersparc  -munaligned-doubles  -mv8

SPARC V9 compilers support the following options
in addition to the above:

-mmedlow  -mmedany
-mint32  -mint64  -mlong32  -mlong64
-mno-stack-bias  -mstack-bias

Convex Options
-mc1  -mc2  -mc32  -mc34  -mc38
-margcount  -mnoargcount
-mlong32  -mlong64
-mvolatile-cache  -mvolatile-nocache

AMD29K Options
-m29000  -m29050  -mbw  -mnbw  -mdw  -mndw
-mlarge  -mnormal  -msmall
-mkernel-registers  -mno-reuse-arg-regs
-mno-stack-check  -mno-storem-bug
-mreuse-arg-regs  -msoft-float  -mstack-check
-mstorem-bug  -muser-registers

ARM Options
-mapcs -m2 -m3 -m6 -mbsd -mxopen -mno-symrename

M88K Options
-m88000  -m88100  -m88110  -mbig-pic
-mcheck-zero-division  -mhandle-large-shift
-midentify-revision  -mno-check-zero-division
-mno-ocs-debug-info  -mno-ocs-frame-position
-mno-optimize-arg-area  -mno-serialize-volatile
-mno-underscores  -mocs-debug-info
```

```
-mocs-frame-position  -moptimize-arg-area
-mserialize-volatile  -mshort-data-num  -msvr3
-msvr4  -mtrap-large-shift  -muse-div-instruction
-mversion-03.00  -mwarn-passed-structs

RS/6000 and PowerPC Options
-mcpu=cpu  type
-mpower  -mno-power  -mpower2  -mno-power2
-mpowerpc  -mno-powerpc
-mpowerpc-gpopt  -mno-powerpc-gpopt
-mpowerpc-gfxopt  -mno-powerpc-gfxopt
-mnew-mnemonics  -mno-new-mnemonics
-mfull-toc   -mminimal-toc  -mno-fop-in-toc  -mno-sum-in-toc
-msoft-float  -mhard-float -mmultiple -mno-multiple
-mstring -mno-string -mbit-align -mno-bit-align
-mstrict-align -mno-strict-align -mrelocatable -mno-relocatable
-mtoc -mno-toc -mtraceback -mno-traceback
-mlittle -mlittle-endian -mbig -mbig-endian

RT Options
-mcall-lib-mul  -mfp-arg-in-fpregs  -mfp-arg-in-gregs
-mfull-fp-blocks  -mhc-struct-return  -min-line-mul
-mminimum-fp-blocks  -mnohc-struct-return

MIPS Options
-mabicalls  -mcpu=cpu  type  -membedded-data
-membedded-pic  -mfp32  -mfp64  -mgas  -mgp32  -mgp64
-mgpopt  -mhalf-pic  -mhard-float  -mint64 -mips1
-mips2 -mips3  -mlong64  -mlong-calls  -mmemcpy
-mmips-as  -mmips-tfile  -mno-abicalls
-mno-embedded-data  -mno-embedded-pic
-mno-gpopt  -mno-long-calls
-mno-memcpy  -mno-mips-tfile  -mno-rnames  -mno-stats
-mrnames -msoft-float
-m4650 -msingle-float -mmad
-mstats  -EL  -EB  -G num  -nocpp

i386 Options
```

```
-m486  -m386 -mieee-fp  -mno-fancy-math-387
-mno-fp-ret-in-387  -msoft-float  -msvr3-shlib
-mno-wide-multiply -mrtd -malign-double
-mreg-alloc=list -mregparm=num
-malign-jumps=num -malign-loops=num
-malign-functions=num

HPPA Options
-mdisable-fpregs  -mdisable-indexing  -mfast-indirect-calls
-mgas  -mjump-in-delay -mlong-millicode-calls -mno-disable-fpregs
-mno-disable-indexing -mno-fast-indirect-calls -mno-gas
-mno-jump-in-delay -mno-millicode-long-calls
-mno-portable-runtime -mno-soft-float -msoft-float
-mpa-risc-1-0  -mpa-risc-1-1  -mportable-runtime -mschedule=list

Intel 960 Options
-mcpu type  -masm-compat  -mclean-linkage
-mcode-align  -mcomplex-addr  -mleaf-procedures
-mic-compat  -mic2.0-compat  -mic3.0-compat
-mintel-asm  -mno-clean-linkage  -mno-code-align
-mno-complex-addr  -mno-leaf-procedures
-mno-old-align  -mno-strict-align  -mno-tail-call
-mnumerics  -mold-align  -msoft-float  -mstrict-align
-mtail-call

DEC Alpha Options
-mfp-regs  -mno-fp-regs  -mno-soft-float
-msoft-float

Clipper Options
-mc300 -mc400

H8/300 Options
-mrelax  -mh

System V Options
-Qy  -Qn  -YP,paths  -Ym,dir
```

```
Code Generation Options
       See section Options for Code Generation Conventions.

       -fcall-saved-reg  -fcall-used-reg
       -ffixed-reg  -finhibit-size-directive
       -fno-common  -fno-ident  -fno-gnu-linker
       -fpcc-struct-return  -fpic  -fPIC
       -freg-struct-return  -fshared-data  -fshort-enums
       -fshort-double  -fvolatile  -fvolatile-global
       -fverbose-asm -fpack-struct +e0  +e1
```

Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of compilation is done:

```
file.c
       C source code which must be preprocessed.

file.i
       C source code which should not be preprocessed.

file.ii
       C++ source code which should not be preprocessed.

file.m
       Objective-C source code. Note that you must link with the library 'l

file.h
       C header file (not to be compiled or linked).

file.cc
file.cxx
file.cpp
file.C
       C++ source code which must be preprocessed. Note that in '.cxx', the last tw
       refers to a literal capital C.

file.s
       Assembler code.

file.S
       Assembler code which must be preprocessed.
```

other An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.

You can specify the input language explicitly with the '-x' option:

```
-x language
       Specify explicitly the language for the following input files (rather than l
       suffix). This option applies to all following input files until the next '-x

       c  objective-c  c++
       c-header  cpp-output  c++-cpp-output
       assembler  assembler-with-cpp

-x none
       Turn off any specification of a language, so that subsequent files are handl
       has not been used at all).
```

If you only want some of the stages of compilation, you can use '-x' (or filename suffixes) to tell gcc where to start, and one of the options '-c', '-S', or '-E' to say where gcc is to stop. Note that some combinations (for example, '-x cpp-output -E' instruct gcc to do nothing at all.

```
-c Compile or assemble the source files, but do not link. The linking stage simply
       object file for each source file.

       By default, the object file name for a source file is made by replacing the
       Unrecognized input files, not requiring compilation or assembly, are

-S Stop after the stage of compilation proper; do not assemble. The output
       non-assembler input file specified.

       By default, the assembler file name for a source file is made by rep

       Input files that don't require compilation are ignored.

-E Stop after the preprocessing stage; do not run the compiler proper. The
       sent to the standard output.

       Input files which don't require preprocessing are ignored.

-o file
       Place output in file file. This applies regardless to whatever sort
       object file, an assembler file or preprocessed C code.

       Since only one output file can be specified, it does not make sense
       are producing an executable file as output.

       If '-o' is not specified, the default is to put an executable file i
       its assembler file in 'source.s', and all preprocessed C source on s

-v Print (on standard error output) the commands executed to run the stages
       compiler driver program and of the preprocessor and the compiler pro

-pipe Use pipes rather than temporary files for communication between the v
       systems where the assembler is unable to read from a pipe; but the G
```

Compiling C++ Programs

C++ source files conventionally use one of the suffixes '.C', '.cc', 'cpp', or '.cxx'; preprocessed C++ files use the suffix '.ii'. GNU CC recognizes files with these names and compiles them as C++ programs even if you call the compiler the same way as for compiling C programs (usually with the name gcc).

However, C++ programs often require class libraries as well as a compiler that understands the C++ language–and under some circumstances, you might want to compile programs from standard input, or otherwise without a suffix that flags them as C++ programs. g++ is a program that calls GNU CC with the default language set to C++, and automatically specifies linking against the GNU class library libg++. (1) On many systems, the script g++ is also installed with the name c++.

When you compile C++ programs, you may specify many of the same command-line options that you use for compiling programs in any language; or command-line options meaningful for C and related languages; or options that are meaningful only for C++ programs. See section Options Controlling C Dialect, for explanations of options for languages related to C. See section Options Controlling C++ Dialect, for explanations of options that are meaningful only for C++ programs.

Options Controlling C Dialect

The following options control the dialect of C (or languages derived from C, such as C++ and Objective C) that the compiler accepts:

```
-ansi Support all ANSI standard C programs.

       This turns off certain features of GNU C that are incompatible with ANSI C,
       predefined macros such as unix and vax that identify the type of system you
       used ANSI trigraph feature, and disallows '$' as part of identifiers.

       The alternate keywords __asm__, __extension__, __inline__ and __typeof__ co
       not want to use them in an ANSI C program, of course, but it is useful to p
       compilations done with '-ansi'. Alternate predefined macros such as __unix_
       '-ansi'.

       The '-ansi' option does not cause non-ANSI programs to be rejected gratuito
       '-ansi'. See section Options to Request or Suppress Warnings.

       The macro __STRICT_ANSI__ is predefined when the '-ansi' option is used. So
       from declaring certain functions or defining certain macros that the ANSI s
       any programs that might use these names for other things.

       The functions alloca, abort, exit, and _exit are not builtin functions when
```

```
-fno-asm
       Do not recognize asm, inline or typeof as a keyword, so that code ca
       __asm__, __inline__ and __typeof__ instead. '-ansi' implies '-fno-as

       In C++, this switch only affects the typeof keyword, since asm and i
       '-fno-gnu-keywords' flag instead, as it also disables the other, C++

-fno-builtin
       Don't recognize builtin functions that do not begin with two leading
       abs, alloca, cos, exit, fabs, ffs, labs, memcmp, memcpy, sin, sqrt,

       GCC normally generates special code to handle certain builtin functi
       single instructions that adjust the stack directly, and calls to mem
       smaller and faster, but since the function calls no longer appear as
       change the behavior of the functions by linking with a different lib

       The '-ansi' option prevents alloca and ffs from being builtin functi
       meaning.

-trigraphs
       Support ANSI C trigraphs. You don't want to know about this brain-da

-traditional
       Attempt to support some aspects of traditional C compilers. Specific

                    All extern declarations take effect globally even if th
              implicit declarations of functions.

                    The newer keywords typeof, inline, signed, const and vo
              alternative keywords such as __typeof__, __inline__, and so o

                    Comparisons between pointers and integers are always al

                    Integer types unsigned short and unsigned char promote

                    Out-of-range floating point literals are not an error.
```

Certain constructs which ANSI regards as a single invalid prep
expressions instead.

String ''constants'' are not necessarily constant; they are st
allocated separately. (This is the same as the effect of '-fwritable

All automatic variables not declared register are preserved by
automatic variables not declared volatile may be clobbered.

The character escape sequences '\x' and '\a' evaluate as the l
'-traditional', '\x' is a prefix for the hexadecimal representation

In C++ programs, assignment to this is permitted with '-tradit
also has this effect.)

You may wish to use '-fno-builtin' as well as '-traditional' if your progra
functions for other purposes of its own.

You cannot use '-traditional' if you include any header files that rely on
systems with ANSI C header files and you cannot use '-traditional' on such
headers.

In the preprocessor, comments convert to nothing at all, rather than to a s

In preprocessing directive, the '#' symbol must appear as the first charact

In the preprocessor, macro arguments are recognized within string constants
though without additional quote marks, when they appear in such a context).
end at a newline.

The predefined macro __STDC__ is not defined when you use '-traditional', b
__GNUC__ indicates are not affected by '-traditional'). If you need to writ
whether '-traditional' is in use, by testing both of these predefined macro
traditional GNU C, other ANSI C compilers, and other old C compilers. The p
defined when you use '-traditional'. See section 'Standard Predefined Macro
these and other predefined macros.

The preprocessor considers a string constant to end at a newline (unless th

'-traditional', string constants can contain the newline character a

-traditional-cpp
: Attempt to support some aspects of traditional C preprocessors. This none of the other effects of '-traditional'.

-fcond-mismatch
: Allow conditional expressions with mismatched types in the second an

-funsigned-char
: Let the type char be unsigned, like unsigned char.

  Each kind of machine has a default for what char should be. It is ei default.

  Ideally, a portable program should always use signed char or unsigne But many programs have been written to use plain char and expect it machines they were written for. This option, and its inverse, let yo

  The type char is always a distinct type from each of signed char or one of those two.

-fsigned-char
: Let the type char be signed, like signed char.

  Note that this is equivalent to '-fno-unsigned-char', which is the n '-fno-signed-char' is equivalent to '-funsigned-char'.

-fsigned-bitfields
-funsigned-bitfields
-fno-signed-bitfields
-fno-unsigned-bitfields
: These options control whether a bitfield is signed or unsigned, when default, such a bitfield is signed, because this is consistent: the

  However, when '-traditional' is used, bitfields are all unsigned no

-fwritable-strings

```
       Store string constants in the writable data segment and don't uniquize them
       assume they can write into string constants. The option '-traditional' also

       Writing into string constants is a very bad idea; ''constants'' should be co

-fallow-single-precision
       Do not promote single precision math operations to double precision, even wh

       Traditional K&R C promotes all floating point operations to double precision
       architecture for which you are compiling, single precision may be faster tha
       but want to use single precision operations when the operands are single pre
       compiling with ANSI or GNU C conventions (the default).
```

Options Controlling C++ Dialect

This section describes the command-line options that are only meaningful for C++ programs; but you can also use most of the GNU compiler options regardless of what language your program is in. For example, you might compile a file firstClass.C like this:

g++ -g -felide-constructors -O -c firstClass.C

In this example, only '-felide-constructors' is an option meant only for C++ programs; you can use the other options with any language supported by GNU CC.

Here is a list of options that are only for compiling C++ programs:

```
-fno-access-control
       Turn off all access checking. This switch is mainly useful for working arou

-fall-virtual
       Treat all possible member functions as virtual, implicitly. All member funct
       member operators) are treated as virtual functions of the class where they a

       This does not mean that all calls to these member functions will be made thr
       circumstances, the compiler can determine that a call to a given virtual fun
       in any case.

-fcheck-new
```

Check that the pointer returned by operator new is non-null before a
Paper requires that operator new never return a null pointer, so thi

-fconserve-space

Put uninitialized or runtime-initialized global variables into the c
cost of not diagnosing duplicate definitions. If you compile with th
completed, you may have an object that is being destroyed twice beca

-fdollars-in-identifiers

Accept '$' in identifiers. You can also explicitly prohibit use of '
C++ allows '$' by default on some target systems but not others.) Tr
identifiers. However, ANSI C and C++ forbid '$' in identifiers.

-fenum-int-equiv

Anachronistically permit implicit conversion of int to enumeration t
the other way around.

-fexternal-templates

Cause template instantiations to obey '#pragma interface' and 'imple
according to the location of the template definition. See section Wh

-falt-external-templates

Similar to -fexternal-templates, but template instances are emitted
section Where's the Template?, for more information.

-fno-gnu-keywords

Do not recognize classof, headof, signature, sigof or typeof as a ke
You can use the keywords __classof__, __headof__, __signature__, __s
'-fno-gnu-keywords'.

-fno-implicit-templates

Never emit code for templates which are instantiated implicitly (i.e
Where's the Template?, for more information.

-fhandle-signatures

Recognize the signature and sigof keywords for specifying abstract t
recognize them. See section Type Abstraction using Signatures.

-fhuge-objects
       Support virtual function calls for objects that exceed the size representab
       default; if you need to use it, the compiler will tell you so. If you compi
       your code with this flag (including libg++, if you use it).

       This flag is not useful when compiling with -fvtable-thunks.

-fno-implement-inlines
       To save space, do not emit out-of-line copies of inline functions controlle
       errors if these functions are not inlined everywhere they are called.

-fmemoize-lookups
-fsave-memoized
       Use heuristics to compile faster. These heuristics are not enabled by defau
       input files compile more slowly.

       The first time the compiler must build a call to a member function (or refe
       class implements member functions of that name; (2) resolve which member fu
       of type conversions need to be made); and (3) check the visibility of the m
       compilation. Normally, the second time a call is made to that member functi
       the same lengthy process again. This means that code like this:

```
cout << ''This '' << p << '' has '' << n << '' legs.\n'';
```

       makes six passes through all three steps. By using a software cache, a ''hi
       cache introduces another layer of mechanisms which must be implemented, and
       enables the software cache.

       Because access privileges (visibility) to members and member functions may
       need to flush the cache. With the '-fmemoize-lookups' flag, the cache is fl
       '-fsave-memoized' flag enables the same software cache, but when the compil
       compiled would yield the same access privileges of the next function to com
       defining many member functions for the same class: with the exception of me
       member function has exactly the same access privileges as every other, and

       The code that implements these flags has rotted; you should probably avoid

-fstrict-prototype

Within an 'extern ''C''' linkage specification, treat a function dec
the function to take no arguments. Normally, such a declaration mean
as in C. '-pedantic' implies '-fstrict-prototype' unless overridden

This flag no longer affects declarations with C++ linkage.

-fno-nonnull-objects
Don't assume that a reference is initialized to refer to a valid obj
references, some old code may rely on them, and you can use '-fno-no

At the moment, the compiler only does this checking for conversions

-foperator-names
Recognize the operator name keywords and, bitand, bitor, compl, not,
'-ansi' implies '-foperator-names'.

-fthis-is-variable
Permit assignment to this. The incorporation of user-defined free st
anachronism. Therefore, by default it is invalid to assign to this w
a member function of class X as a non-lvalue of type 'X *'. However,
'-fthis-is-variable'.

-fvtable-thunks
Use 'thunks' to implement the virtual function dispatch table ('vtab
vtables was to store a pointer to the function and two offsets for a
store a single pointer to a 'thunk' function which does any necessar

This option also enables a heuristic for controlling emission of vta
be emitted in the translation unit containing the first one of those

-nostdinc++
Do not search for header files in the standard directories specific
option is used when building libg++.)

-traditional
For C++ programs (in addition to the effects that apply to both C an
See section Options Controlling C Dialect.

```
In addition, these optimization, warning, and code generation options have meaning

-fno-default-inline
       Do not assume 'inline' for functions defined inside a class scope. See secti

-Wenum-clash
-Woverloaded-virtual
-Wtemplate-debugging
       Warnings that apply only to C++ programs. See section Options to Request or

+en Control how virtual function definitions are used, in a fashion compatible wit
       Conventions.
```

Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error.

You can request many specific warnings with options beginning '-W', for example '-Wimplicit' to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning '-Wno-' to turn off warnings; for example, '-Wno-implicit'. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of warnings produced by GNU CC:

```
-fsyntax-only
       Check the code for syntax errors, but don't do anything beyond that.

-pedantic
       Issue all the warnings demanded by strict ANSI standard C; reject all progr

       Valid ANSI standard C programs should compile properly with or without this
       However, without this option, certain GNU extensions and traditional C feat
       rejected.

       '-pedantic' does not cause warning messages for use of the alternate keywor
       warnings are also disabled in the expression that follows __extension__. Ho
```

escape routes; application programs should avoid them. See section A

This option is not intended to be useful; it exists only to satisfy
the ANSI standard.

Some users try to use '-pedantic' to check programs for strict ANSI
they want: it finds some non-ANSI practices, but not all--only those

A feature to report any failure to conform to ANSI C might be useful
work and would be quite different from '-pedantic'. We recommend, ra
C and disregard the limitations of other compilers. Aside from certa
less reason ever to use any other C compiler other than for bootstra

-pedantic-errors
Like '-pedantic', except that errors are produced rather than warnin

-w Inhibit all warning messages.

-Wno-import
Inhibit warning messages about the use of '#import'.

-Wchar-subscripts
Warn if an array subscript has type char. This is a common cause of
some machines.

-Wcomment
Warn whenever a comment-start sequence '/*' appears in a comment.

-Wformat
Check calls to printf and scanf, etc., to make sure that the argumen
specified.

-Wimplicit
Warn whenever a function or parameter is implicitly declared.

-Wparentheses
Warn if parentheses are omitted in certain contexts, such as when th
or when operators are nested whose precedence people often get confu

```
-Wreturn-type
      Warn whenever a function is defined with a return-type that defaults to int
      return-value in a function whose return-type is not void.

-Wswitch
      Warn whenever a switch statement has an index of enumeral type and lacks a c
      enumeration. (The presence of a default label prevents this warning.) case l
      warnings when this option is used.

-Wtrigraphs
      Warn if any trigraphs are encountered (assuming they are enabled).

-Wunused
      Warn whenever a variable is unused aside from its declaration, whenever a fu
      label is declared but not used, and whenever a statement computes a result t

      To suppress this warning for an expression, simply cast it to void. For unus
      (see section Specifying Attributes of Variables).

-Wuninitialized
      An automatic variable is used without first being initialized.

      These warnings are possible only in optimizing compilation, because they req
      optimizing. If you don't specify '-O', you simply won't get these warnings.

      These warnings occur only for variables that are candidates for register all
      declared volatile, or whose address is taken, or whose size is other than 1
      unions or arrays, even when they are in registers.

      Note that there may be no warning about a variable that is used only to comp
      computations may be deleted by data flow analysis before the warnings are pr

      These warnings are made optional because GNU CC is not smart enough to see a
      appearing to have an error. Here is one example of how this can happen:

      {
        int x;
```

```
  switch (y)
    {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
    }
  foo (x);
}
```

If the value of y is always 1, 2 or 3, then x is always initialized,

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  ...
  if (change_y) y = save_y;
}
```

This has no bug because save_y is used only if it is set.

Some spurious warnings can be avoided if you declare all the functio
Attributes of Functions.

-Wenum-clash
: Warn about conversion between different enumeration types. (C++ only

-Wreorder (C++ only)
: Warn when the order of member initializers given in the code does no

```
struct A {
  int i;
  int j;
  A(): j (0), i (1) { }
};
```

Here the compiler will warn that the member initializers for 'i' and members.

-Wtemplate-debugging
When using templates in a C++ program, warn if debugging is not yet fully a

-Wall All of the above '-W' options combined. These are all the options which pert
believe is easy to avoid, even in conjunction with macros.

The remaining '-W...' options are not implied by '-Wall' because they warn about c
occasion, in clean programs.

-W Print extra warning messages for these events:

A nonvolatile automatic variable might be changed by a call to
in optimizing compilation.

The compiler sees only the calls to setjmp. It cannot know where lon
it at any point in the code. As a result, you may get a warning even
cannot in fact be called at the place which would cause a problem.

A function can return either with or without a value. (Falling
returning without a value.) For example, this function would evoke s

```
foo (a)
{
  if (a > 0)
    return a;
}
```

An expression-statement contains no side effects.

An unsigned value is compared against zero with '<' or '<='.

A comparison like 'x<=y<=z' appears; this is equivalent to '(x
interpretation from that of ordinary mathematical notation.

Storage-class specifiers like static are not the first things
is obsolescent.

```
                  If '-Wall' or '-Wunused' is also specified, warn about

                  An aggregate has a partly bracketed initializer. For ex
            because braces are missing around the initializer for x.h:

            struct s { int f, g; };
            struct t { struct s h; int i; };
            struct t x = { 1, 2, 3 };

-Wtraditional
      Warn about certain constructs that behave differently in traditional

                  Macro arguments occurring within string constants in th
            traditional C, but are part of the constant in ANSI C.

                  A function declared external in one block and then used

                  A switch statement has an operand of type long.

-Wshadow
      Warn whenever a local variable shadows another local variable.

-Wid-clash-len
      Warn whenever two distinct identifiers match in the first len charac
      certain obsolete, brain-damaged compilers.

-Wlarger-than-len
      Warn whenever an object of larger than len bytes is defined.

-Wpointer-arith
      Warn about anything that depends on the ''size of'' a function type
      convenience in calculations with void * pointers and pointers to fun

-Wbad-function-cast
      Warn whenever a function call is cast to a non-matching type. For ex

-Wcast-qual
```

Warn whenever a pointer is cast so as to remove a type qualifier from the ta
to an ordinary char *.

-Wcast-align

Warn whenever a pointer is cast such that the required alignment of the targ
an int * on machines where integers can only be accessed at two- or four-by

-Wwrite-strings

Give string constants the type const char[length] so that copying the addres
warning. These warnings will help you find at compile time code that can try
been very careful about using const in declarations and prototypes. Otherwis
make '-Wall' request these warnings.

-Wconversion

Warn if a prototype causes a type conversion that is different from what wou
prototype. This includes conversions of fixed point to floating and vice ver
a fixed point argument except when the same as the default promotion.

Also, warn if a negative integer constant expression is implicitly convertec
assignment x = -1 if x is unsigned. But do not warn about explicit casts lil

-Waggregate-return

Warn if any functions that return structures or unions are defined or callec
elicits a warning.)

-Wstrict-prototypes

Warn if a function is declared or defined without specifying the argument ty
without a warning if preceded by a declaration which specifies the argument

-Wmissing-prototypes

Warn if a global function is defined without a previous prototype declaratio
provides a prototype. The aim is to detect global functions that fail to be

-Wmissing-declarations

Warn if a global function is defined without a previous declaration. Do so e
option to detect global functions that are not declared in header files.

-Wredundant-decls

Warn if anything is declared more than once in the same scope, even
nothing.

-Wnested-externs
Warn if an extern declaration is encountered within an function.

-Winline
Warn if a function can not be inlined, and either it was declared as

-Woverloaded-virtual
Warn when a derived class function declaration may be an error in de
definitions of virtual functions must match the type signature of a
compiler warns when you define a function with the same name as a vi
any declarations from the base class.

-Wsynth (C++ only)
Warn when g++'s synthesis behavior does not match that of cfront. Fo

```
struct A {
  operator int ();
  A& operator = (int);
};

main ()
{
  A a,b;
  a = b;
}
```

In this example, g++ will synthesize a default 'A& operator = (const
='.

-Werror
Make all warnings into errors.

Options for Debugging Your Program or GNU CC

GNU CC has various special options that are used for debugging either your program or GCC:

```
-g Produce debugging information in the operating system's native format (stabs, C
       this debugging information.

       On most systems that use stabs format, '-g' enables use of extra debugging
       information makes debugging work better in GDB but will probably make other
       you want to control for certain whether to generate the extra information,
       '-gdwarf+', or '-gdwarf' (see below).

       Unlike most other C compilers, GNU CC allows you to use '-g' with '-O'. The
       occasionally produce surprising results: some variables you declared may no
       you did not expect it; some statements may not be executed because they com
       hand; some statements may execute in different places because they were mov

       Nevertheless it proves possible to debug optimized output. This makes it re
       have bugs.

       The following options are useful when GNU CC is generated with the capabili

-ggdb Produce debugging information in the native format (if that is supported), i

-gstabs
       Produce debugging information in stabs format (if that is supported), witho
       most BSD systems. On MIPS, Alpha and System V Release 4 systems this option
       understood by DBX or SDB. On System V Release 4 systems this option require

-gstabs+
       Produce debugging information in stabs format (if that is supported), using
       (GDB). The use of these extensions is likely to make other debuggers crash

-gcoff Produce debugging information in COFF format (if that is supported). This i
       prior to System V Release 4.

-gxcoff
       Produce debugging information in XCOFF format (if that is supported). This
       RS/6000 systems.

-gxcoff+
       Produce debugging information in XCOFF format (if that is supported)
       debugger (GDB). The use of these extensions is likely to make other
       assemblers other than the GNU assembler (GAS) to fail with an error.

-gdwarf
       Produce debugging information in DWARF format (if that is supported)
       4 systems.

-gdwarf+
       Produce debugging information in DWARF format (if that is supported)
       debugger (GDB). The use of these extensions is likely to make other

-glevel
-ggdblevel
-gstabslevel
-gcofflevel
-gxcofflevel
-gdwarflevel
       Request debugging information and also use level to specify how much

       Level 1 produces minimal information, enough for making backtraces i
       includes descriptions of functions and external variables, but no in

       Level 3 includes extra information, such as all the macro definition
       expansion when you use '-g3'.

-p Generate extra code to write profile information suitable for the analys
       the source files you want data about, and you must also use it when

-pg Generate extra code to write profile information suitable for the analy
       the source files you want data about, and you must also use it when

-a Generate extra code to write profile information for basic blocks, which
       the basic block start address, and the function name containing the
       start of the basic block will also be recorded. If not overridden by
```

file 'bb.out'.

This data could be analyzed by a program like tcov. Note, however, that the
GNU gprof should be extended to process this data.

-dletters
Says to make debugging dumps during compilation at times specified by lette
names for most of the dumps are made by appending a word to the source file
are the possible letters for use in letters, and their meanings:

'M' Dump all macro definitions, at the end of preprocessing, and write no o
'N' Dump all macro names, at the end of preprocessing.
'D' Dump all macro definitions, at the end of preprocessing, in addition to
'y' Dump debugging information during parsing, to standard error.
'r' Dump after RTL generation, to 'file.rtl'.
'x' Just generate RTL for a function instead of compiling it. Usually used
'j' Dump after first jump optimization, to 'file.jump'.
's' Dump after CSE (including the jump optimization that sometimes follows
'L' Dump after loop optimization, to 'file.loop'.
't' Dump after the second CSE pass (including the jump optimization that so
'f' Dump after flow analysis, to 'file.flow'.
'c' Dump after instruction combination, to the file 'file.combine'.
'S' Dump after the first instruction scheduling pass, to 'file.sched'.
'l' Dump after local register allocation, to 'file.lreg'.
'g' Dump after global register allocation, to 'file.greg'.
'R' Dump after the second instruction scheduling pass, to 'file.sched2'.
'J' Dump after last jump optimization, to 'file.jump2'.
'd' Dump after delayed branch scheduling, to 'file.dbr'.
'k' Dump after conversion from registers to stack, to 'file.stack'.
'a' Produce all the dumps listed above.
'm' Print statistics on memory usage, at the end of the run, to standard er
'p' Annotate the assembler output with a comment indicating which pattern a

-fpretend-float When running a cross-compiler, pretend that the target mach
machine. This causes incorrect output of the actual floating constants, but
as GNU CC would make when running on the target machine.

-save-temps Store the usual ''temporary'' intermediate files permanently; p

```
        the source file. Thus, compiling 'foo.c' with '-c -save-temps' would

        -print-file-name=library Print the full absolute name of the library
        anything else. With this option, GNU CC does not compile or link any

        -print-prog-name=program Like '-print-file-name', but searches for a

        -print-libgcc-file-name Same as '-print-file-name=libgcc.a'.

        This is useful when you use '-nostdlib' or '-nodefaultlibs' but you

        gcc -nostdlib files... 'gcc -print-libgcc-file-name'

        -print-search-dirs Print the name of the configured installation dir
        search--and don't do anything else.

        This is useful when gcc prints the error message 'installation probl
        directory'. To resolve this you either need to put 'cpp' and the oth
        you can set the environment variable GCC_EXEC_PREFIX to the director
        section Environment Variables Affecting GNU CC.
```

Options That Control Optimization

These options control various sorts of optimizations:

```
-O
-O1 Optimize. Optimizing compilation takes somewhat more time, and a lot mo

        Without '-O', the compiler's goal is to reduce the cost of compilati
        Statements are independent: if you stop the program with a breakpoin
        variable or change the program counter to any other statement in the
        source code.

        Without '-O', the compiler only allocates variables declared registe
        produced by PCC without '-O'.

        With '-O', the compiler tries to reduce code size and execution time
        When you specify '-O', the compiler turns on '-fthread-jumps' and '-fdefer-p
        '-fdelayed-branch' on machines that have delay slots, and '-fomit-frame-poin
        even without a frame pointer. On some machines the compiler also turns on ot

-O2 Optimize even more. GNU CC performs nearly all supported optimizations that do
        does not perform loop unrolling or function inlining when you specify '-O2'
        compilation time and the performance of the generated code.

        '-O2' turns on all optional optimizations except for loop unrolling and func
        machines where doing so does not interfere with debugging.

-O3 Optimize yet more. '-O3' turns on all optimizations specified by '-O2' and also

-O0 Do not optimize.

        If you use multiple '-O' options, with or without level numbers, the last su
```

Options of the form '-fflag' specify machine-independent flags. Most flags have both positive and negative forms; the negative form of '-ffoo' would be '-fno-foo'. In the table below, only one of the forms is listed–the one which is not the default. You can figure out the other form by either removing 'no-' or adding it.

```
-ffloat-store
        Do not store floating point variables in registers, and inhibit other option
        from a register or memory.

        This option prevents undesirable excess precision on machines such as the 68
        more precision than a double is supposed to have. For most programs, the exc
        rely on the precise definition of IEEE floating point. Use '-ffloat-store' f

-fno-default-inline
        Do not make member functions inline by default merely because they are defin
        when you specify '-O', member functions defined inside class scope are compi
        'inline' in front of the member function name.

-fno-defer-pop
        Always pop the arguments to each function call as soon as that function retu
```

function call, the compiler normally lets arguments accumulate on th

-fforce-mem
: Force memory operands to be copied into registers before doing arith
memory references potential common subexpressions. When they are not
eliminate the separate register-load. I am interested in hearing abo

-fforce-addr
: Force memory address constants to be copied into registers before do
'-fforce-mem' may. I am interested in hearing about the difference t

-fomit-frame-pointer
: Don't keep the frame pointer in a register for functions that don't
frame pointers; it also makes an extra register available in many fu
machines.

On some machines, such as the Vax, this flag has no effect, because
pointer and nothing is saved by pretending it doesn't exist. The mac
whether a target machine supports this flag. See section Register Us

-fno-inline
: Don't pay attention to the inline keyword. Normally this option is u
Note that if you are not optimizing, no functions can be expanded in

-finline-functions
: Integrate all simple functions into their callers. The compiler heur
integrating in this way.

If all calls to a given function are integrated, and the function is
assembler code in its own right.

-fkeep-inline-functions
: Even if all calls to a given function are integrated, and the functi
callable version of the function.

-fno-function-cse
: Do not put function addresses in registers; make each instruction th
explicitly.

```
        This option results in less efficient code, but some strange hacks that alt
        performed when this option is not used.

-ffast-math
        This option allows GCC to violate some ANSI or IEEE rules and/or specificat
        example, it allows the compiler to assume arguments to the sqrt function ar
        are NaNs.

        This option should never be turned on by any '-O' option since it can resul
        exact implementation of IEEE or ANSI rules/specifications for math function
```

The following options control specific optimizations. The '-O2' option turns on all of these optimizations except '-funroll-loops' and '-funroll-all-loops'. On most machines, the '-O' option turns on the '-fthread-jumps' and '-fdelayed-branch' options, but specific machines may handle it differently.

You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

```
-fstrength-reduce
        Perform the optimizations of loop strength reduction and elimination of ite

-fthread-jumps
        Perform optimizations where we check to see if a jump branches to a locatio
        found. If so, the first branch is redirected to either the destination of t
        depending on whether the condition is known to be true or false.

-fcse-follow-jumps
        In common subexpression elimination, scan through jump instructions when th
        For example, when CSE encounters an if statement with an else clause, CSE w
        false.

-fcse-skip-blocks
        This is similar to '-fcse-follow-jumps', but causes CSE to follow jumps whi
        encounters a simple if statement with no else clause, '-fcse-skip-blocks' c
        if.

-frerun-cse-after-loop
```

Re-run common subexpression elimination after loop optimizations has

-fexpensive-optimizations
Perform a number of minor optimizations that are relatively expensiv

-fdelayed-branch
If supported for the target machine, attempt to reorder instructions
instructions.

-fschedule-insns
If supported for the target machine, attempt to reorder instructions
This helps machines that have slow floating point or memory load ins
result of the load or floating point instruction is required.

-fschedule-insns2
Similar to '-fschedule-insns', but requests an additional pass of in
This is especially useful on machines with a relatively small number
one cycle.

-fcaller-saves
Enable values to be allocated in registers that will be clobbered by
registers around such calls. Such allocation is done only when it se

This option is enabled by default on certain machines, usually those

-funroll-loops
Perform the optimization of loop unrolling. This is only done for lo
time or run time. '-funroll-loop' implies both '-fstrength-reduce' a

-funroll-all-loops
Perform the optimization of loop unrolling. This is done for all loo
'-funroll-all-loops' implies '-fstrength-reduce' as well as '-frerun

-fno-peephole
Disable any machine-specific peephole optimizations.

Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.

If you use the '-E' option, nothing is done except preprocessing. Some of these options make sense only together with '-E' because they cause the preprocessor output to be unsuitable for actual compilation.

```
-include file
       Process file as input before processing the regular input file. In effect, t
       options on the command line are always processed before '-include file', reg
       the '-include' and '-imacros' options are processed in the order in which th

-imacros file
       Process file as input, discarding the resulting output, before processing th
       file is discarded, the only effect of '-imacros file' is to make the macros

       Any '-D' and '-U' options on the command line are always processed before '-
       they are written. All the '-include' and '-imacros' options are processed in

-idirafter dir
       Add the directory dir to the second include path. The directories on the sec
       found in any of the directories in the main include path (the one that '-I'

-iprefix prefix
       Specify prefix as the prefix for subsequent '-iwithprefix' options.

-iwithprefix dir
       Add a directory to the second include path. The directory's name is made by
       specified previously with '-iprefix'. If you have not specified a prefix yet
       compiler is used as the default.

-iwithprefixbefore dir
       Add a directory to the main include path. The directory's name is made by co
       '-iwithprefix'.

-isystem dir
       Add a directory to the beginning of the second include path, marking it as a
```

treatment as is applied to the standard system directories.

-nostdinc
Do not search the standard system directories for header files. Only
current directory, if appropriate) are searched. See section Options

By using both '-nostdinc' and '-I-', you can limit the include-file

-undef Do not predefine any nonstandard macros. (Including architecture fla

-E Run only the C preprocessor. Preprocess all the C source files specified
output file.

-C Tell the preprocessor not to discard comments. Used with the '-E' option

-P Tell the preprocessor not to generate '#line' directives. Used with the

-M Tell the preprocessor to output a rule suitable for make describing the
preprocessor outputs one make-rule whose target is the object file n
#include header files it uses. This rule may be a single line or may
printed on standard output instead of the preprocessed C program.

'-M' implies '-E'.

Another way to specify output of a make rule is by setting the envir
Environment Variables Affecting GNU CC).

-MM Like '-M' but the output mentions only the user header files included w
'#include <file>' are omitted.

-MD Like '-M' but the dependency information is written to a file made by r
is in addition to compiling the file as specified---'-MD' does not i

In Mach, you can use the utility md to merge multiple dependency fil
'make' command.

-MMD Like '-MD' except mention only user header files, not system header fi

```
-MG Treat missing header files as generated files and assume they live in the same
       must also specify either '-M' or '-MM'. '-MG' is not supported with '-MD' o

-H Print the name of each header file used, in addition to other normal activities

-Aquestion(answer)
       Assert the answer answer for question, in case it is tested with a preproce
       #question(answer)'. '-A-' disables the standard assertions that normally de

-Dmacro
       Define macro macro with the string '1' as its definition.

-Dmacro=defn
       Define macro macro as defn. All instances of '-D' on the command line are p

-Umacro
       Undefine macro macro. '-U' options are evaluated after all '-D' options, bu

-dM Tell the preprocessor to output only a list of the macro definitions that are
       option.

-dD Tell the preprocessing to pass all macro definitions into the output, in their

-dN Like '-dD' except that the macro arguments and contents are omitted. Only '#de

-trigraphs
       Support ANSI C trigraphs. The '-ansi' option also has this effect.

-Wp,option
       Pass option as an option to the preprocessor. If option contains commas, it
```

Passing Options to the Assembler

You can pass options to the assembler.

```
-Wa,option
       Pass option as an option to the assembler. If option contains commas, it is
```

Options for Linking

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

```
object-file-name
       A file name that does not end in a special recognized suffix is cons
       distinguished from libraries by the linker according to the file con
       linker.

-c
-S
-E If any of these options is used, then the linker is not run, and object
       Controlling the Kind of Output.

-llibrary
       Search the library named library when linking.

       It makes a difference where in the command you write this option; th
       they are specified. Thus, 'foo.o -lz bar.o' searches library 'z' aft
       functions in 'z', those functions may not be loaded.

       The linker searches a standard list of directories for the library,
       uses this file as if it had been specified precisely by name.

       The directories searched include several standard system directories

       Normally the files found this way are library files--archive files w
       scanning through it for members which define symbols that have so fa
       is an ordinary object file, it is linked in the usual fashion. The o
       name is that '-l' surrounds library with 'lib' and '.a' and searches

-lobjc You need this special case of the '-l' option in order to link an Ob

-nostartfiles
       Do not use the standard system startup files when linking. The stand
       -nodefaultlibs is used.
```

-nodefaultlibs
    Do not use the standard system libraries when linking. Only the libraries yo
    startup files are used normally, unless -nostartfiles is used.

-nostdlib
    Do not use the standard system startup files or libraries when linking. No s
    passed to the linker.

    One of the standard libraries bypassed by '-nostdlib' and '-nodefaultlibs' i
    that GNU CC uses to overcome shortcomings of particular machines, or special
    to GNU CC Output, for more discussion of 'libgcc.a'.) In most cases, you nee
    standard libraries. In other words, when you specify '-nostdlib' or '-nodefa
    well. This ensures that you have no unresolved references to internal GNU CC
    to ensure C++ constructors will be called; see section collect2.)

-s Remove all symbol table and relocation information from the executable.

-static
    On systems that support dynamic linking, this prevents linking with the shar

-shared
    Produce a shared object which can then be linked with other objects to form

-symbolic
    Bind references to global symbols when building a shared object. Warn about
    link editor option '-Xlinker -z -Xlinker defs'). Only a few systems support

-Xlinker option
    Pass option as an option to the linker. You can use this to supply system-sp
    how to recognize.

    If you want to pass an option that takes an argument, you must use '-Xlinker
    argument. For example, to pass '-assert definitions', you must write '-Xlink
    does not work to write '-Xlinker ''-assert definitions''', because this pass
    not what the linker expects.

-Wl,option
    Pass option as an option to the linker. If option contains commas, it is spl

```
-u symbol
       Pretend the symbol symbol is undefined, to force linking of library
       different symbols to force loading of additional library modules.
```

Options for Directory Search

These options specify directories to search for header files, for libraries and for parts of the compiler:

```
-Idir Add the directory directory to the head of the list of directories to
       header file, substituting your own version, since these directories
       more than one '-I' option, the directories are scanned in left-to-ri

-I- Any directories you specify with '-I' options before the '-I-' option a
       are not searched for '#include <file>'.

       If additional directories are specified with '-I' options after the
       (Ordinarily all '-I' directories are used this way.)

       In addition, the '-I-' option inhibits the use of the current direct
       directory for '#include ''file'''. There is no way to override this
       directory which was current when the compiler was invoked. That is n
       but it is often satisfactory.

       '-I-' does not inhibit the use of the standard system directories fo

-Ldir Add directory dir to the list of directories to be searched for '-l'.

-Bprefix
       This option specifies where to find the executables, libraries, incl

       The compiler driver program runs one or more of the subprograms 'cpp
       each program it tries to run, both with and without 'machine/version
       Version).

       For each subprogram to be run, the compiler driver first tries the '
       specified, the driver tries two standard prefixes, which are '/usr/l
```

```
        those results in a file name that is found, the unmodified program name is s
        environment variable.

        '-B' prefixes that effectively specify directory names also apply to librar
        options into '-L' options for the linker. They also apply to includes files
        options into '-isystem' options for the preprocessor. In this case, the comp

        The run-time support file 'libgcc.a' can also be searched for using the '-B
        standard prefixes above are tried, and that is all. The file is left out of

        Another way to specify a prefix much like the '-B' prefix is to use the envi
        Environment Variables Affecting GNU CC.
```

Specifying Target Machine and Compiler Version

By default, GNU CC compiles code for the same type of machine that you are using. However, it can also be installed as a cross-compiler, to compile for some other type of machine. In fact, several different configurations of GNU CC, for different target machines, can be installed side by side. Then you specify which one to use with the '-b' option.

In addition, older and newer versions of GNU CC can be installed side by side. One of them (probably the newest) will be the default, but you may sometimes wish to use another.

```
-b machine
        The argument machine specifies the target machine for compilation. This is u
        cross-compiler.

        The value to use for machine is the same as was specified as the machine typ
        example, if a cross-compiler was configured with 'configure i386v', meaning
        you would specify '-b i386v' to run that cross compiler.

        When you do not specify '-b', it normally means to compile for the same type

-V version
        The argument version specifies which version of GNU CC to run. This is usefu
        example, version might be '2.0', meaning to run GNU CC version 2.0.
```

```
The default version, when you do not specify '-V', is the last versi
```

The '-b' and '-V' options actually work by controlling part of the file name used for the executable files and libraries used for compilation. A given version of GNU CC, for a given target machine, is normally kept in the directory '/usr/local/lib/gcc-lib/machine/version'.

Thus, sites can customize the effect of '-b' or '-V' either by changing the names of these directories or adding alternate names (or symbolic links). If in directory '/usr/local/lib/gcc-lib/' the file '80386' is a link to the file 'i386v', then '-b 80386' becomes an alias for '-b i386v'.

In one respect, the '-b' or '-V' do not completely change to a different compiler: the top-level driver program gcc that you originally invoked continues to run and invoke the other executables (preprocessor, compiler per se, assembler and linker) that do the real work. However, since no real work is done in the driver program, it usually does not matter that the driver program in use is not the one for the specified target and version.

The only way that the driver program depends on the target machine is in the parsing and handling of special machine-specific options. However, this is controlled by a file which is found, along with the other executables, in the directory for the specified version and target machine. As a result, a single installed driver program adapts to any specified target machine and compiler version.

The driver program executable does control one significant thing, however: the default version and target machine. Therefore, you can install different instances of the driver program, compiled for different targets or versions, under different names.

For example, if the driver for version 2.0 is installed as ogcc and that for version 2.1 is installed as gcc, then the command gcc will use version 2.1 by default, while ogcc will use 2.0 by default. However, you can choose either version with either command with the '-V' option.

Environment Variables Affecting GNU CC

This section describes several environment variables that affect how GNU CC operates. They work by specifying directories or prefixes to use when searching for various kinds of files.

Note that you can also specify places to search using options such as '-B', '-I' and '-L' (see section Options for Directory Search). These take precedence over places specified using environment variables, which in turn take precedence over those specified by the configuration of GNU CC. See section Controlling the Compilation Driver, 'gcc'.

```
TMPDIR If TMPDIR is set, it specifies the directory to use for temporary files. GNU
       compilation which is to be used as input to the next stage: for example, the
       compiler proper.

GCC_EXEC_PREFIX
       If GCC_EXEC_PREFIX is set, it specifies a prefix to use in the names of the
       when this prefix is combined with the name of a subprogram, but you can spec

       If GNU CC cannot find the subprogram using the specified prefix, it tries lo

       The default value of GCC_EXEC_PREFIX is 'prefix/lib/gcc-lib/' where prefix i
       'configure' script.

       Other prefixes specified with '-B' take precedence over this prefix.

       This prefix is also used for finding files such as 'crt0.o' that are used fo

       In addition, the prefix is used in an unusual way in finding the directories
       directories whose name normally begins with '/usr/local/lib/gcc-lib' (more p
       GNU CC tries replacing that beginning with the specified prefix to produce a
       CC will search 'foo/bar' where it would normally search '/usr/local/lib/bar
       the standard directories come next.

COMPILER_PATH
       The value of COMPILER_PATH is a colon-separated list of directories, much li
       when searching for subprograms, if it can't find the subprograms using GCC_E

LIBRARY_PATH
       The value of LIBRARY_PATH is a colon-separated list of directories, much lik
       CC tries the directories thus specified when searching for special linker fi
       using GNU CC also uses these directories when searching for ordinary librari
       '-L' come first).

C_INCLUDE_PATH
```

```
CPLUS_INCLUDE_PATH
OBJC_INCLUDE_PATH
       These environment variables pertain to particular languages. Each va
       PATH. When GNU CC searches for header files, it tries the directorie
       directories specified with '-I' but before the standard header file

DEPENDENCIES_OUTPUT
       If this variable is set, its value specifies how to output dependenc
       This output looks much like the output from the '-M' option (see sec
       separate file, and is in addition to the usual results of compilatio

       The value of DEPENDENCIES_OUTPUT can be just a file name, in which c
       name from the source file name. Or the value can have the form 'file
       using target as the target name.
```

Running Protoize

The program protoize is an optional part of GNU C. You can use it to add prototypes to a program, thus converting the program to ANSI C in one respect. The companion program unprotoize does the reverse: it removes argument types from any prototypes that are found.

When you run these programs, you must specify a set of source files as command line arguments. The conversion programs start out by compiling these files to see what functions they define. The information gathered about a file foo is saved in a file named 'foo.X'.

After scanning comes actual conversion. The specified files are all eligible to be converted; any files they include (whether sources or just headers) are eligible as well.

But not all the eligible files are converted. By default, protoize and unprotoize convert only source and header files in the current directory. You can specify additional directories whose files should be converted with the '-d directory' option. You can also specify particular files to exclude with the '-x file' option. A file is converted if it is eligible, its directory name matches one of the specified directory names, and its name within the directory has not been excluded.

Basic conversion with protoize consists of rewriting most function definitions and function declarations to specify the types of the arguments. The only ones not rewritten are those for varargs functions.

protoize optionally inserts prototype declarations at the beginning of the source file, to make them available for any calls that precede the function's definition. Or it can insert prototype declarations with block scope in the blocks where undeclared functions are called.

Basic conversion with unprotoize consists of rewriting most function declarations to remove any argument types, and rewriting function definitions to the old-style pre-ANSI form.

Both conversion programs print a warning for any function declaration or definition that they can't convert. You can suppress these warnings with '-q'.

The output from protoize or unprotoize replaces the original source file. The original file is renamed to a name ending with '.save'. If the '.save' file already exists, then the source file is simply discarded.

protoize and unprotoize both depend on GNU CC itself to scan the program and collect information about the functions it uses. So neither of these programs will work until GNU CC is installed.

Here is a table of the options you can use with protoize and unprotoize. Each option works with both programs unless otherwise stated.

```
-B directory
       Look for the file 'SYSCALLS.c.X' in directory, instead of the usual directo
       prototype information about standard system functions. This option applies 

-c compilation-options
       Use compilation-options as the options when running gcc to produce the '.X'
       passed in addition, to tell gcc to write a '.X' file.

       Note that the compilation options must be given as a single argument to pro
       gcc options, you must quote the entire set of compilation options to make t

       There are certain gcc arguments that you cannot use, because they would pro
       '-O', '-c', '-S', and '-o' If you include these in the compilation-options,

-C Rename files to end in '.C' instead of '.c'. This is convenient if you are conv
       to protoize.

-g Add explicit global declarations. This means inserting explicit declarations at
       is called in the file and was not declared. These declarations precede the 
       function. This option applies only to protoize.

-i string
       Indent old-style parameter declarations with the string string. This

       unprotoize converts prototyped function definitions to old-style fun
       argument list and the initial '{'. By default, unprotoize uses five
       space instead, use '-i '' '''.

-k Keep the '.X' files. Normally, they are deleted after conversion is fini

-l Add explicit local declarations. protoize with '-l' inserts a prototype
       function without any declaration. This option applies only to protoi

-n Make no real changes. This mode just prints information about the conver

-N Make no '.save' files. The original files are simply deleted. Use this o

-p program
       Use the program program as the compiler. Normally, the name 'gcc' is

-q Work quietly. Most warnings are suppressed.

-v Print the version number, just like '-v' for gcc.
```

A.3 Extensions to the C Language Family

GNU C provides several language features not found in ANSI standard C. (The '-pedantic' option directs GNU CC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro `__GNUC__`, which is always defined under GNU CC.

These extensions are available in C and Objective C. Most of them are also available in C++. See section Extensions to the C++ Language, for extensions that apply only to C++.

Statements and Declarations in Expressions

A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of foo ().

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type void, and thus effectively no value.)

This feature is especially useful in making macro definitions ”safe” (so that they evaluate each operand exactly once). For example, the ”maximum” function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either a or b twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here let’s assume int), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit field, or the initial value of a static variable.

If you don’t know the type of the operand, you can still do this, but you must use typeof (see section Referring to a Type with typeof) or type naming (see section Naming an Expression’s Type).

Locally Declared Labels

Each statement expression is a scope in which local labels can be declared. A local label is simply an identifier; you can jump to it with an ordinary goto statement, but only from within the statement expression it belongs to.

A local label declaration looks like this:

```
__label__ label;
```

or

```
__label__ label1, label2, ...;
```

Local label declarations must come at the beginning of the statement expression, right after the `'({'`, before any ordinary declarations.

The label declaration defines the label name, but does not define the label itself. You must do this in the usual way, with label:, within the statements of the statement expression.

The local label feature is useful because statement expressions are often used in macros. If the macro contains nested loops, a goto can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function. A local label avoids this problem. For example:

```
#define SEARCH(array, target)                     \
({                                               \
  __label__ found;                                \
  typeof (target) _SEARCH_target = (target);      \
  typeof (*(array)) *_SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
      if (_SEARCH_array[i][j] == _SEARCH_target)  \
        { value = i; goto found; }              \
  value = -1;                                     \
 found:                                           \
  value;                                          \
})
```

Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator '&&'. The value has type void *. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement, `goto *exp;`. For example,

```
goto *ptr;
```

Any expression of type `void *` is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

Note that this does not check whether the subscript is in bounds — array indexing in C never does that.

Such an array of label values serves a purpose much like that of the switch statement. The switch statement is cleaner, so use that rather than an array unless the problem does not fit a switch statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

You can use this mechanism to jump to code in a different function. If you do that, totally unpredictable things will happen. The best way to avoid this is to store the label address only in automatic variables and never pass it as an argument.

Constructing Function Calls

Using the built-in functions described below, you can record the arguments a function received, and call another function with the same arguments, without knowing the number or types of the arguments.

You can also record the return value of that function call, and later return that value, without knowing what data type the function tried to return (as long as your caller expects that data type).

```
__builtin_apply_args ()
       This built-in function returns a pointer of type void * to data desc
       passed to the current function.

       The function saves the arg pointer register, structure value address
       function into a block of memory allocated on the stack. Then it retu

__builtin_apply (function, arguments, size)
       This built-in function invokes function (type void (*)()) with a cop
       *) and size (type int).

       The value of arguments should be the value returned by __builtin_app
       stack argument data, in bytes.

       This function returns a pointer of type void * to data describing ho
       data is saved in a block of memory allocated on the stack.

       It is not always simple to compute the proper value for size. The va
       data that should be pushed on the stack and copied from the incoming

__builtin_return (result)
       This built-in function returns the value described by result from th
       returned by __builtin_apply.
```

Naming an Expression's Type

You can give a name to the type of an expression using a typedef declaration with an initializer. Here is how to define name as a type name for the type of exp:

```
typedef name = exp;

This is useful in conjunction with the statements-within-expressions feature. Here
''maximum'' macro that operates on any arithmetic type:

#define max(a,b) \
  ({typedef _ta = (a), _tb = (b);  \
    _ta _a = (a); _tb _b = (b);     \
    _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for a and b. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

Referring to a Type with typeof

Another way to refer to the type of an expression is with typeof. The syntax of using of this keyword looks like sizeof, but the construct acts semantically like a type name defined with typedef.

There are two ways of writing the argument to typeof: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that x is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to int.

If you are writing a header file that must work when included in ANSI C programs, write `__typeof__` instead of typeof. See section Alternate Keywords.

A typeof-construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of sizeof or typeof.

This declares y with the type of what x points to.

```
typeof (*x) y;
```

This declares y as an array of such values.

```
typeof (*x) y[4];
```

This declares y as an array of pointers to characters:

```
typeof (typeof (char *)[4]) y;
```

It is equivalent to the following traditional C declaration:

```
char *y[4];
```

To see the meaning of the declaration using typeof, and why it might be a useful way to write, let's rewrite it with these macros:

```
#define pointer(T)  typeof(T *)
#define array(T, N) typeof(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, array (pointer (char), 4) is the type of arrays of 4 pointers to char.

Generalized Lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them.

Standard C++ allows compound expressions and conditional expressions as lvalues, and permits casts to reference type, so use of this extension is deprecated for C++ code.

For example, a compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

```
(a, b) += 5
a, (b += 5)
```

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

```
&(a, b)
a, &b
```

A conditional expression is a valid lvalue if its type is not void and the true and false branches are both valid lvalues. For example, these two expressions are equivalent:

```
(a ? b : c) = 5
(a ? b = 5 : (c = 5))
```

A cast is a valid lvalue if its operand is an lvalue. A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if a has type char *, the following two expressions are equivalent:

```
(int)a = 5
(int)(a = (char *)(int)5)
```

An assignment-with-arithmetic operation such as '+=' applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:

```
(int)a += 5
(int)(a = (char *)(int) ((int)a + 5))
```

You cannot take the address of an lvalue cast, because the use of its address would not work out coherently. Suppose that `&(int)f` were permitted, where f has type float. Then the following statement would try to store an integer bit-pattern where a floating point number belongs:

```
*&(int)f = 1;
```

This is quite different from what `(int)f = 1` would do — that would convert 1 to floating point and store it. Rather than cause this inconsistency, we think it is better to prohibit use of '&' on a cast.

If you really do want an `int *` pointer with the address of `f`, you can simply write `(int *)&f`.

Conditionals with Omitted Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ? : y
```

has the value of x if that is nonzero; otherwise, the value of y.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

Double-Word Integers

GNU C supports data types for integers that are twice as long as long int. Simply write long long int for a signed integer, or unsigned long long int for an unsigned integer. To make an integer constant of type long long int, add the suffix LL to the integer. To make an integer constant of type unsigned long long int, add the suffix ULL to the integer.

You can use these types in arithmetic like any other integer types. Addition, subtraction, and bitwise boolean operations on these types are open-coded on all types of machines. Multiplication is open-coded if the machine supports fullword-to-doubleword a widening multiply instruction. Division and shifts are open-coded only on machines that provide special support. The operations that are not open-coded use special library routines that come with GNU CC.

There may be pitfalls when you use long long types for function arguments, unless you declare function prototypes. If a function expects type int for its argument, and you pass a value of type long long int, confusion will result because the caller and the subroutine will disagree about the number of bytes for the argument. Likewise, if the function expects long long int and you pass int. The best way to avoid such problems is to use prototypes.

subsubsectionComplex Numbers

GNU C supports complex data types. You can declare both complex integer types and complex floating types, using the keyword `__complex__`.

For example,

```
__complex__ double x;
```

declares x as a variable whose real part and imaginary part are both of type double.

```
__complex__ short int y;
```

declares y to have real and imaginary parts of type short int; this is not likely to be useful, but it shows that the set of complex types is complete.

To write a constant with a complex data type, use the suffix 'i' or 'j' (either one; they are equivalent). For example, 2.5fi has type `__complex__` float and 3i has type `__complex__` int. Such a constant always has a pure imaginary value, but you can form any complex value you like by adding one to a real constant.

To extract the real part of a complex-valued expression exp, write `__real__ exp`. Likewise, use `__imag__` to extract the imaginary part.

The operator '∼' performs complex conjugation when used on a value with a complex type.

GNU CC can allocate complex automatic variables in a noncontiguous fashion; it's even possible for the real part to be in a register while the imaginary part is on the stack (or vice-versa). None of the supported debugging info formats has a way to represent noncontiguous allocation like this, so GNU CC describes a noncontiguous complex variable as if it were two separate variables of noncomplex type. If the variable's actual name is foo, the two fictitious variables are named foo$realandfoo$imag. You can examine and set these two fictitious variables with your debugger.

A future version of GDB will know how to recognize such pairs and treat them as a single variable with a complex type.

Arrays of Length Zero

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};

{
  struct line *thisline = (struct line *)
    malloc (sizeof (struct line) + this_length);
  thisline->length = this_length;
}
```

In standard C, you would have to give contents a length of 1, which means either you waste space or complicate the argument to malloc.

Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace-level is exited. For example:

```
FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; you get an error message for it.

You can use the function alloca to get an effect much like variable-length arrays. The function alloca is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with alloca exists until the containing function returns. The space for a variable-length array is deallocated as soon as the array name's scope ends. (If you use both variable-length arrays and alloca in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with alloca.)

You can also use variable-length arrays as arguments to functions:

```
struct entry
tester (int len, char data[len][len])
{
  ...
}
```

The length of an array is computed once when the storage is allocated and is remembered for the scope of the array in case you access it with sizeof.

If you want to pass the array first and the length afterward, you can use a forward declaration in the parameter list–another GNU extension.

```
struct entry
tester (int len; char data[len][len], int len)
```

```
{
  ...
}
```

The 'int len' before the semicolon is a parameter forward declaration, and it serves the purpose of making the name len known when the declaration of data is parsed.

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a "real" declaration in parameter name and data type.

Macros with Variable Numbers of Arguments

In GNU C, a macro can accept a variable number of arguments, much as a function can. The syntax for defining the macro looks much like that used for a function. Here is an example:

```
#define eprintf(format, args...)  \
 fprintf (stderr, format , ## args)
```

Here args is a rest argument: it takes in zero or more arguments, as many as the call contains. All of them plus the commas between them form the value of args, which is substituted into the macro body where args is used. Thus, we have this expansion:

```
eprintf (''%s:%d: '', input_file_name, line_number)
==>
fprintf (stderr, ''%s:%d: '' , input_file_name, line_number)
```

Note that the comma after the string constant comes from the definition of eprintf, whereas the last comma comes from the value of args.

The reason for using '##' is to handle the case when args matches no arguments at all. In this case, args has an empty value. In this case, the second comma in the definition becomes an embarrassment: if it got through to the expansion of the macro, we would get something like this:

```
fprintf (stderr, ''success!\n'' , );
```

which is invalid C syntax. '##' gets rid of the comma, so we get the following instead:

```
fprintf (stderr, ''success!\n'');
```

This is a special feature of the GNU C preprocessor: '##' before a rest argument that is empty discards the preceding sequence of non-whitespace characters from the macro definition. (If another macro argument precedes, none of it is discarded.)

It might be better to discard the last preprocessor token instead of the last preceding sequence of non-whitespace characters; in fact, we may someday change this feature to do so. We advise you to write the macro definition so that the preceding sequence of non-whitespace characters is just a single token, so that the meaning will not change if we change the definition of this feature.

Non-Lvalue Arrays May Have Subscripts

Subscripting is allowed on arrays that are not lvalues, even though the unary '&' operator is not. For example, this is valid in GNU C though not valid in other C dialects:

```
struct foo {int a[4];};

struct foo f();

bar (int index)
{
  return f().a[index];
}
```

Arithmetic on void- and Function-Pointers

In GNU C, addition and subtraction operations are supported on pointers to void and on pointers to functions. This is done by treating the size of a void or of a function as 1.

A consequence of this is that sizeof is also allowed on void and on function types, and returns 1.

The option '-Wpointer-arith' requests a warning if these extensions are used.

Non-Constant Initializers

As in standard C++, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  ...
}
```

Constructor Expressions

GNU C supports constructor expressions. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer.

Usually, the specified type is a structure. Assume that struct foo and structure are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a struct foo with a constructor:

```
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```
{
  struct foo temp = {x + y, 'a', 0};
  structure = temp;
}
```

You can also construct an array. If all the elements of the constructor are (made up of) simple constant expressions, suitable for use in initializers, then the constructor is an lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { ''x'', ''y'', ''z'' };
```

Array constructors whose elements are not simple constants are not very useful, because the constructor is not an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a switch statement, while the latter does the same thing an ordinary C initializer would do. Here is an example of subscripting an array constructor:

```
output = ((int[]) { 2, x, 28 }) [input];
```

Constructor expressions for scalar types and union types are is also allowed, but then the constructor expression is equivalent to a cast.

Labeled Elements in Initializers

Standard C requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized.

In GNU C you can give the elements in any order, specifying the array indices or structure field names they apply to. This extension is not implemented in GNU C++.

To specify an array index, write '[index]' or '[index] =' before the element value. For example,

```
int a[6] = { [4] 29, [2] = 15 };
```

is equivalent to

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

The index values must be constant expressions, even if the array being initialized is automatic.

To initialize a range of elements to the same value, write '[first ... last] = value'. For example,

```
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```

Note that the length of the array is the highest value specified plus one.

In a structure initializer, specify the name of a field to initialize with 'fieldname:' before the element value. For example, given the following structure,

```
struct point { int x, y; };
```

the following initialization

```
struct point p = { y: yvalue, x: xvalue };
```

is equivalent to

```
struct point p = { xvalue, yvalue };
```

Another syntax which has the same meaning is '.fieldname ='., as shown here:

```
struct point p = { .y = yvalue, .x = xvalue };
```

You can also use an element label (with either the colon syntax or the period-equal syntax) when initializing a union, to specify which element of the union should be used. For example,

```
union foo { int i; double d; };

union foo f = { d: 4 };
```

will convert 4 to a double to store it in the union using the second element. By contrast, casting 4 to type union foo would store it into the union as the integer i, since it is an integer. (See section Cast to a Union Type.)

You can combine this technique of naming elements with ordinary C initialization of successive elements. Each initializer element that does not have a label applies to the next consecutive element of the array or structure. For example,

```
int a[6] = { [1] = v1, v2, [4] = v4 };
```

is equivalent to

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an enum type. For example:

```
int whitespace[256]
  = { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
      ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

Case Ranges

You can specify a range of consecutive values in a single case label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual case labels, one for each integer value from low to high, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the ..., for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

Cast to a Union Type

A cast to union type is similar to other casts, except that the type specified is a union type. You can specify the type either with union tag or with a typedef name. A cast to union is actually a constructor though, not a cast, and hence does not yield an lvalue like normal casts. (See section Constructor Expressions.)

The types that may be cast to the union type are those of the members of the union. Thus, given the following union and variables:

```
union foo { int i; double d; };
int x;
double y;
```

both x and y can be cast to type union foo.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union:

```
union foo u;
...
u = (union foo) x  ==  u.i = x
u = (union foo) y  ==  u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
...
hack ((union foo) x);
```

Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword `__attribute__` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. Eight attributes, noreturn, const, format, section, constructor, destructor, unused and weak are currently defined for functions. Other attributes, including section are supported for variables declarations (see section Specifying Attributes of Variables) and for types (see section Specifying Attributes of Types).

You may also specify attributes with '__' preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use `__noreturn__` instead of noreturn.

noreturn

A few standard library functions, such as abort and exit, cannot return. GNU CC knows this automatically. Some programs define their own functions that never return. You can declare them noreturn to tell the compiler this fact. For example,

```
void fatal () __attribute__ ((noreturn));

void
fatal (...)
{
  ... /* Print error message. */ ...
```

```
      exit (1);
    }
```

The noreturn keyword tells the compiler to assume that fatal cannot return. It can then optimize without regard to what would happen if fatal ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

Do not assume that registers saved by the calling function are restored before calling the noreturn function.

It does not make sense for a noreturn function to have a return type other than void.

The attribute noreturn is not implemented in GNU C versions earlier than 2.5. An alternative way to declare that a function does not return, which works in the current version and in some older versions, is as follows:

```
typedef void voidfn ();

volatile voidfn fatal;
```

const

Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute const. For example,

```
int square (int) __attribute__ ((const));
```

says that the hypothetical function square is safe to call fewer times than the program says.

The attribute const is not implemented in GNU C versions earlier than 2.5. An alternative way to declare that a function has no side effects, which works in the current version and in some older versions, is as follows:

```
typedef int intfn ();

extern const intfn square;
```

This approach does not work in GNU C++ from 2.6.0 on, since the language specifies that the 'const' must be attached to the return value.

Note that a function that has pointer arguments and examines the data pointed to must not be declared const. Likewise, a function that calls a non-const function usually must not be const. It does not make sense for a const function to return void.

format `(archetype, string-index, first-to-check)`

The format attribute specifies that a function takes printf or scanf style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
      __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to `my_printf` for consistency with the printf style format string argument `my_format`.

The parameter archetype determines how the format string is interpreted, and should be either printf or scanf. The parameter string-index specifies which argument is the format string argument (starting from 1), while first-to-check is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as vprintf), specify the third parameter as zero. In this case the compiler only checks the format string for consistency.

In the example above, the format string (`my_format`) is the second argument of the function `my_print`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The format attribute allows you to identify your own functions which take format strings as arguments, so that GNU CC can check the calls to these functions for errors. The compiler always checks formats for the ANSI library functions printf, fprintf, sprintf, scanf, fscanf, sscanf, vprintf, vfprintf and vsprintf whenever such warnings are requested (using '-Wformat'), so there is no need to modify the header file 'stdio.h'.

section `(''section-name'')`

Normally, the compiler places the code it generates in the text section. Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The section attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__ ((section (''bar'')));
```

puts the function foobar in the bar section.

Some file formats do not support arbitrary sections so the section attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

constructor/destructor

The constructor attribute causes the function to be called automatically before execution enters `main ()`. Similarly, the destructor attribute causes the function to be called automatically after `main ()` has completed or `exit ()` has been called. Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

These attributes are not currently implemented for Objective C.

unused

This attribute, attached to a function, means that the function is meant to be possibly unused. GNU CC will not produce a warning for this function.

weak

The weak attribute causes the declaration to be emitted as a weak symbol rather than a global. This is primarily useful in defining library functions which can be overridden in user code, though it can also be used with non-function declarations. Weak symbols are supported for ELF targets, and also for a.out targets when using the GNU assembler and linker.

alias `(''target'')`

The alias attribute causes the declaration to be emitted as an alias for another symbol, which must be specified. For instance,

```
void __f () { /* do something */; }
void f () __attribute__ ((weak, alias (''__f'')));
```

declares 'f' to be a weak alias for '`__f`'. In C++, the mangled name for the target must be used.

You can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following an attribute declaration with another attribute declaration.

Some people object to the `__attribute__` feature, suggesting that ANSI C's `#pragma` should be used instead. There are two reasons for not doing this.

- It is impossible to generate `#pragma` commands from a macro.
- .There is no telling what the same `#pragma` might mean in another compiler.

These two reasons apply to almost any application that might be proposed for `#pragma`. It is basically a mistake to use `#pragma` for anything.

Prototypes and Old-Style Function Definitions

GNU C extends ANSI C to allow a function prototype to override a later old-style non-prototype definition. Consider the following example:

```
/* Use prototypes unless the compiler is old-fashioned.  */
#if __STDC__
#define P(x) x
#else
#define P(x) ()
#endif

/* Prototype function declaration.  */
int isroot P((uid_t));

/* Old-style function definition.  */
int
isroot (x)   /* ??? lossage here ??? */
     uid_t x;
{
  return x == 0;
}
```

Suppose the `type uid_t` happens to be short. ANSI C does not allow this example, because subword arguments in old-style non-prototype definitions are promoted. Therefore in this example the function definition's argument is really an int, which does not match the prototype argument type of short.

This restriction of ANSI C makes it hard to write code that is portable to traditional C compilers, because the programmer does not know whether the `uid_t` type is short, int, or long. Therefore, in cases like these GNU C allows a prototype to override a later old-style definition. More precisely, in GNU

C, a function prototype argument type overrides the argument type specified by a later old-style definition if the former type is the same as the latter type before promotion. Thus in GNU C the above example is equivalent to the following:

```
int isroot (uid_t);

int
isroot (uid_t x)
{
  return x == 0;
}
```

Note: GNU C++ does not support old-style function definitions, so this extension is irrelevant.

Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers.

On some machines, dollar signs are allowed in identifiers if you specify '-traditional'. On a few systems they are allowed by default, even if you do not use '-traditional'. But they are never allowed if you specify '-ansi'.

There are certain ANSI C programs (obscure, to be sure) that would compile incorrectly if dollar signs were permitted in identifiers. For example:

```
#define foo(a) #a
#define lose(b) foo (b)
#define test$
lose (test)
```

The Character ESC in Constants

You can use the sequence '\e' in a string or character constant to stand for the ASCII character `ESC`.

Inquiring on Alignment of Types or Variables

The keyword `__alignof__` allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like sizeof.

For example, if the target machine requires a double value to be aligned on an 8-byte boundary, then `__alignof__` (double) is 8. This is true on many RISC machines. On more traditional machine designs, `__alignof__` (double) is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd addresses. For these machines, `__alignof__` reports the recommended alignment of a type.

When the operand of `__alignof__`is an lvalue rather than a type, the value is the largest alignment that the lvalue is known to have. It may have this alignment as a result of its data type, or because it is part of a structure and inherits alignment from that structure. For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of `__alignof__` (foo1.y) is probably 2 or 4, the same as `__alignof__` (int), even though the data type of foo1.y does not itself demand any alignment.

A related feature which lets you specify the alignment of an object is `__alignof__`((aligned (alignment))); see the following section.

Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Eight attributes are currently defined for variables: aligned, mode, nocommon, packed, section, transparent_union, unused, and weak. Other attributes are available for functions (see section Declaring Attributes of Functions) and for types (see section Specifying Attributes of Types).

You may also specify attributes with '__' preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use `__aligned__` instead of aligned.

aligned (alignment)

This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable x on a 16-byte boundary. On a 68040, this could be used in conjunction with an asm expression to access the move16 instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned int pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a double member that forces the union to be double-word aligned.

It is not possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed. You cannot specify alignment for a typedef name because such a name is just an alias, not a distinct type.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The aligned attribute can only increase the alignment; but you can decrease it by specifying packed as well. See below.

Note that the effectiveness of aligned attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying aligned(16) in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

mode (mode)

This attribute specifies the data type for the declaration–whichever type corresponds to the mode mode. This in effect lets you request an integer or floating point type according to its width.

You may also specify a mode of '`byte`' or '`__byte__`' to indicate the mode corresponding to a one-byte integer, 'word' or '`__word__`' for the mode of a one-word integer, and '`pointer`' or '`__pointer__`' for the mode used to represent pointers.

nocommon

This attribute specifies requests GNU CC not to place a variable "common" but instead to allocate space for it directly. If you specify the '-fno-common' flag, GNU CC will do this for all variables.

Specifying the nocommon attribute for a variable provides an initialization of zeros. A variable may only be initialized in one source file.

packed

The packed attribute specifies that a variable or structure field should have the smallest possible alignment–one byte for a variable, and one bit for a field, unless you specify a larger value with the aligned attribute.

Here is a structure in which the field x is packed, so that it immediately follows a:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

section ("section-name")

Normally, the compiler places the objects it generates in sections like data and bss. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The section attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section (''DUART_A''))) = { 0 };
struct duart b __attribute__ ((section (''DUART_B''))) = { 0 };
char stack[10000] __attribute__ ((section (''STACK''))) = { 0 };
int init_data_copy __attribute__ ((section (''INITDATACOPY''))) = 0;

main()
{
  /* Initialize stack pointer */
  init_sp (stack + sizeof (stack));

  /* Initialize initialized data */
  memcpy (&init_data_copy, &data, &edata - &data);

  /* Turn on the serial ports */
  init_duart (&a);
  init_duart (&b);
}
```

Use the section attribute with an initialized definition of a global variable, as shown in the example. GNU CC issues a warning and otherwise ignores the section attribute in uninitialized variable declarations.

You may only use the section attribute with a fully initialized global definition because of the way linkers work. The linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the common (or bss) section and can be multiply "defined". You can force a variable to be initialized with the '-fno-common' flag or the nocommon attribute.

Some file formats do not support arbitrary sections so the section attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

transparent_union

This attribute, attached to a function argument variable which is a union, means to pass the argument in the same way that the first union member would be passed. You can also use this attribute on a typedef for a union data type; then it applies to all function arguments with that type.

unused

This attribute, attached to a variable, means that the variable is meant to be possibly unused. GNU CC will not produce a warning for this variable.

weak

The weak attribute is described in See section Declaring Attributes of Functions.

To specify multiple attributes, separate them by commas within the double parentheses: for example,

```
__attribute__ ((aligned (16), packed))
```

Specifying Attributes of Types

The keyword `__attribute__` allows you to specify special attributes of struct and union types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Three attributes are currently defined for types: aligned, packed, and transparent_union. Other attributes are defined for functions (see section Declaring Attributes of Functions) and for variables (see section Specifying Attributes of Variables).

You may also specify any one of these attributes with '__' preceding and following its keyword. This allows you to use these attributes in header files without being concerned about a possible macro of the same name. For example, you may use `__aligned__` instead of aligned.

You may specify the aligned and transparent_union attributes either in a typedef declaration or just past the closing curly brace of a complete enum, struct or union type definition and the packed attribute only past the closing brace of a definition.

aligned (alignment)

This attribute specifies a minimum alignment (in bytes) for variables of the specified type. For example, the declarations:

```
    struct S { short f[3]; } __attribute__ ((aligned (8));
    typedef int more_aligned_int __attribute__ ((aligned (8));
```

force the compiler to insure (as fas as it can) that each variable whose type is struct S or `more_aligned_int` will be allocated and aligned at least on a 8-byte boundary. On a Sparc, having all variables of type struct S aligned to 8-byte boundaries allows the compiler to use the ldd and std (doubleword load and store) instructions when copying one variable of type struct S to another, thus improving run-time efficiency.

Note that the alignment of any given struct or union type is required by the ANSI C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the struct or union in question. This means that you can effectively adjust the alignment of a struct or union type by attaching an aligned attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire struct or union type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given struct or union type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

In the example above, if the size of each short is 2 bytes, then the size of the entire struct S type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire struct S type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations will often be more efficient for efficiently-aligned types than for other types.

The aligned attribute can only increase the alignment; but you can decrease it by specifying packed as well. See below.

Note that the effectiveness of aligned attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying aligned(16) in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

packed

This attribute, attached to an enum, struct, or union type definition, specified that the minimum required memory be used to represent the type.

Specifying this attribute for struct and union types is equivalent to specifying the packed attribute on each of the structure or union members. Specifying the '-fshort-enums' flag on the line is equivalent to specifying the packed attribute on all enum definitions.

You may only specify this attribute after a closing curly brace on an enum definition, not in a typedef declaration.

transparent_union

This attribute, attached to a union type definition, indicates that any variable having that union type should, if passed to a function, be passed in the same way that the first union member would be passed. For example:

```
union foo
{
  char a;
  int x[2];
} __attribute__ ((transparent_union));
```

To specify multiple attributes, separate them by commas within the double parentheses: for example,

```
__attribute__ ((aligned (16), packed))
```

An Inline Function is As Fast As a Macro

By declaring a function inline, you can direct GNU CC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions is an optimization and it really "works" only in optimizing compilation. If you don't use '-O', no function is really inline.

To declare a function inline, use the inline keyword in its declaration, like this:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(If you are writing a header file to be included in ANSI C programs, write `__inline__` instead of inline. See section Alternate Keywords.)

You can also make all "simple enough" functions inline with the option '-finline-functions'. Note that certain usages in a function definition can make it unsuitable for inline substitution.

Note that in C and Objective C, unlike C++, the inline keyword does not affect the linkage of the function.

GNU CC automatically inlines member functions defined within the class body of C++ programs even if they are not explicitly declared inline. (You can override this with '-fno-default-inline'; see section Options Controlling C++ Dialect.)

When a function is both inline and static, if all calls to the function are integrated into the caller, and the function's address is never used, then the function's own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the option '-fkeep-inline-functions'. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined.

When an inline function is not static, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-static inline function is always compiled on its own in the usual fashion.

If you specify both inline and extern in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of inline and extern has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking inline and extern) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

GNU C does not inline any functions when not optimizing. It is not clear whether it is better to inline or not, in this case, but we found that a correct implementation when not optimizing was difficult. So we did the easy thing, and turned it off.

Alternate Keywords

The option '-traditional' disables certain keywords; '-ansi' disables certain others. This causes trouble when you want to use GNU C extensions, or ANSI C features, in a general-purpose header file that should be usable by all programs, including ANSI C programs and traditional ones. The keywords asm, typeof and inline cannot be used since they won't work in a program compiled with '-ansi', while the keywords const, volatile, signed, typeof and inline won't work in a program compiled with '-traditional'.

The way to solve these problems is to put '__' at the beginning and end of each problematical keyword. For example, use `__asm__` instead of asm, `__const__` instead of const, and `__inline__` instead of inline.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

'-pedantic' causes warnings for many GNU C extensions. You can prevent such warnings within one expression by writing ˆ_extension__ before the expression. `__extension__` has no effect aside from this.

Incomplete enum Types

You can define an enum tag without specifying its possible values. This results in an incomplete type, much like what you get if you write struct foo without describing the elements. A later declaration which does specify the possible values completes the type.

You can't allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of enum more consistent with the way struct and union are handled.

This extension is not supported by GNU C++.

Function Names as Strings

GNU CC predefines two string variables to be the name of the current function. The variable `__FUNCTION__` is the name of the function as it appears in the source. The variable `__PRETTY_FUNCTION__` is the name of the function pretty printed in a language specific fashion.

These names are always the same in a C function, but in a C++ function they may be different. For example, this program:

```
extern ''C'' {
extern int printf (char *, ...);
}

class a {
 public:
  sub (int i)
    {
      printf (''__FUNCTION__ = %s\n'', __FUNCTION__);
      printf (''__PRETTY_FUNCTION__ = %s\n'', __PRETTY_FUNCTION__);
    }
};
```

```
int
main (void)
{
  a ax;
  ax.sub (0);
  return 0;
}
```

gives this output:

```
__FUNCTION__ = sub
__PRETTY_FUNCTION__ = int  a::sub (int)
```