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.

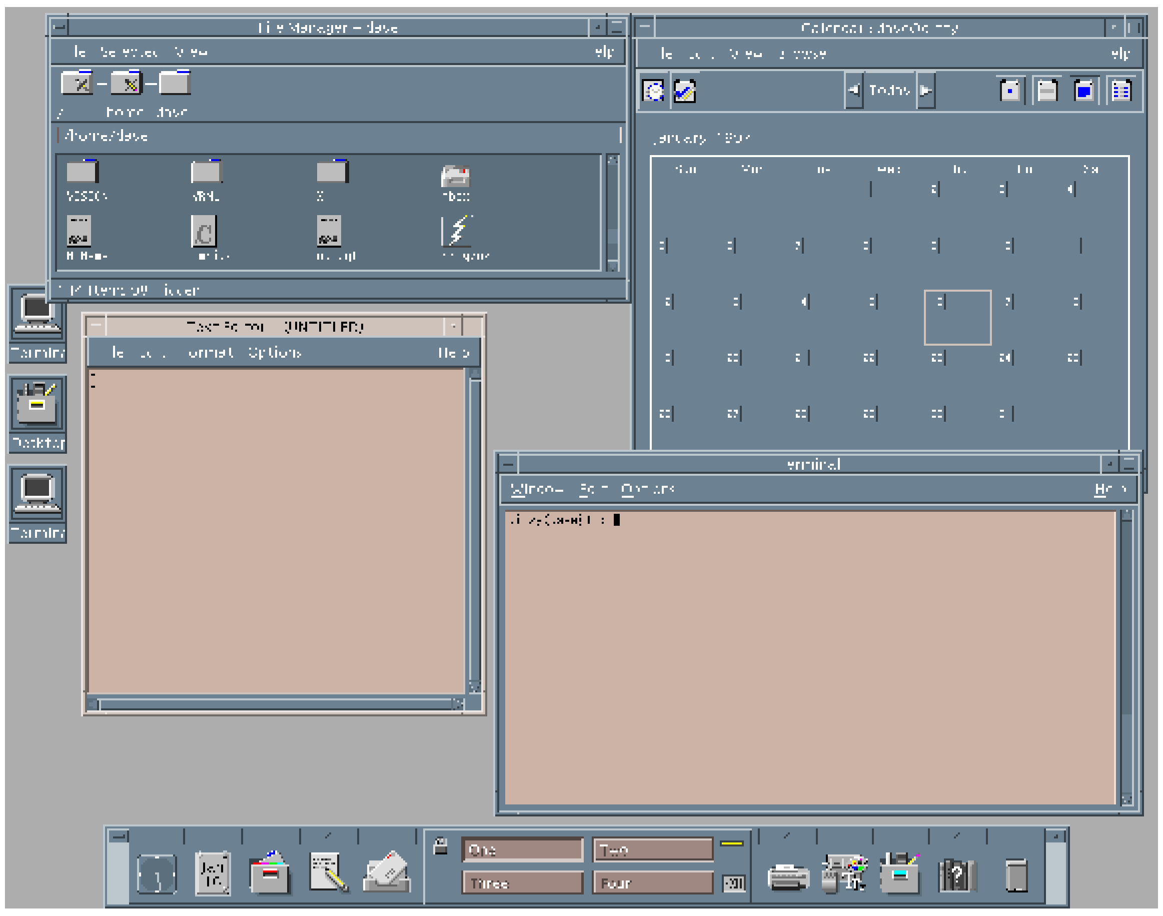

Fig. 1.1 Sample CDE Desktop

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 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.

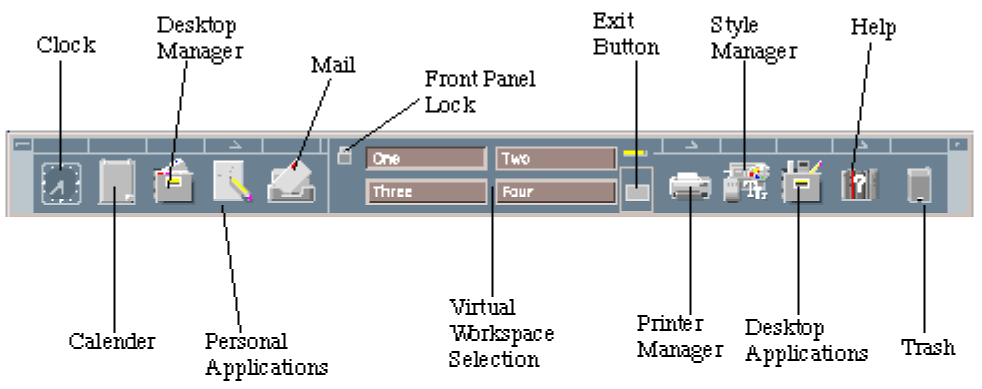

Fig. 1.2 Clients, Servers and Xlib

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.

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.

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.

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.

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.

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.

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:

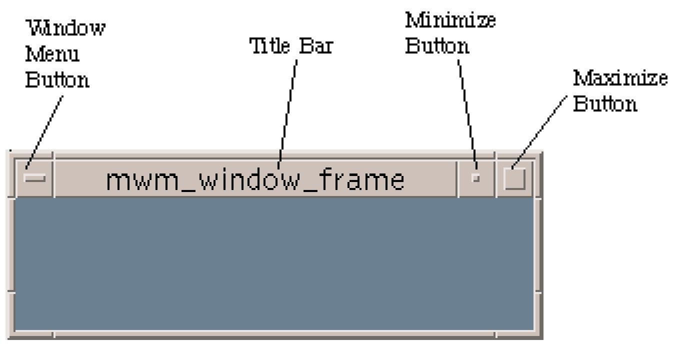

Fig. 1.3 The Motif Window Frame

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 ❑).

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

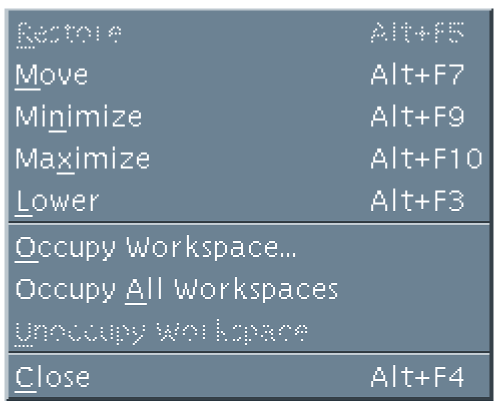

Fig. 1.4 The Window Menu

- **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 ❑) provide no ***internal*** means of termination. The `Close` option being the only means to achieve this.

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

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:

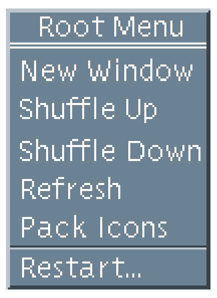

Fig. 1.6 The *mwm* Root Menu

Fig. 1.7 The CDE *dtwm* Root Menu

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.
Restart	-- Restart the Workspace.
Logout	(***dtwm***) -- Quit the Window Manager.

Exercises

Exercise 12158

Exercise~\ref{ex.cde1}

Add an application to the application manager

Exercise 12159

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 12160

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 12161

Find out how to resize windows etc. and practice this.

Exercise 12162

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 12163

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 b called new.

3. Delete the file spare.

4. Display your original file on the terminal.

5. Print your file out.

Exercise 12164

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

---

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.

Creating, Compiling and Running Your Program

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

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).

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 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 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 <u>more</u> 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 .

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.

The C Compilation Model

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

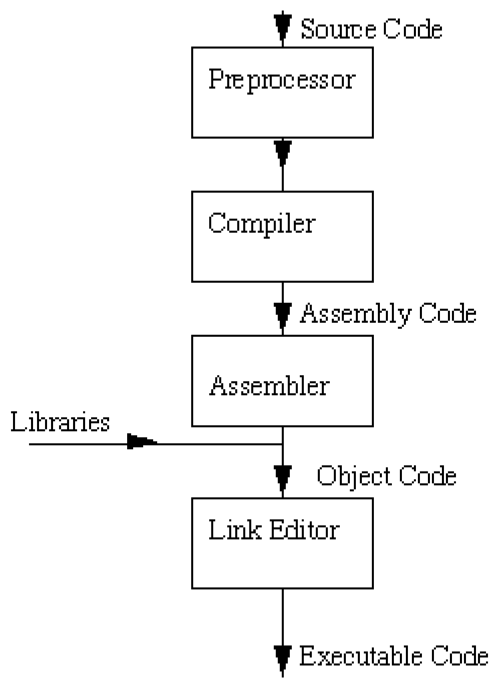

Fig. 2.1 The C Compilation Model

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`

C Compiler

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

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.

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).

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 ❑ 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 ❏.

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.

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.

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().
```

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`).

Exercises

Exercise 12171

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 12172

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 12173

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 12174

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 12175

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 12176

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

Figure 1.5: Sample Icon from Xterm Application

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`?

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.

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''.

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. Alternatively you may run it from here and see the output.

Clearly nobody ever writes code like or at least should never. This piece of code actually one an international Obfuscated C Code Contest 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).

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
```

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 `ints` are `long ints` 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;
```

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 real float;
                typedef letter char;
```

Variables declared:

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

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).

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.

Arithmetic Operations

As well as the standard arithmetic operators (`+ - * /`) 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 **NOT** *x* = *x***y* + 2.

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.

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).

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
                );
```

Exercises

Write C programs to perform the following tasks.

Exercise 12270

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

Exercise 12271

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

Exercise 12272

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

Exercise 12273

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 12274

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 12275

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 12276

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 12277

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 12278

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 12279

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.

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.

The `if` statement

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

```
  if  (expression)
                              statement
```

...or:

```
  if  (expression)
                              statement1
            else
                              statement2
```

...or:

```
  if  (expression)
                              statement1
            else if (expression)
                              statement2
            else
                              statement3
```

For example:-

```
  int x,y,w;

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

                               }
```

The `?` operator

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

```
  expression1 ? expression2:  expression3
```

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;
```

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 item1:
                                              statement1;
                                              break;
                               case item2:
                                              statement2;
                                              break;
                                              ⋮                                            ⋮
                                              statementn;
                                              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.`

Exercises

Exercise 12304

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 12305

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

Exercise 12306

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 12307

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);

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.

The `for` statement

The C `for` statement has the following form:

```
  for  (expression1; 2; expression3)
                                 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;

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).

The `do-while` statement

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

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

It is similar to PASCAL's `repeat` ... `until` <u>except</u> `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.`

`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 */
```

Exercises

Exercise 12327

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

Exercise 12328

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 12329

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 12330

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 12331

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 12332

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 12333

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.